Temporal-Numeric Planning with Control Parameters

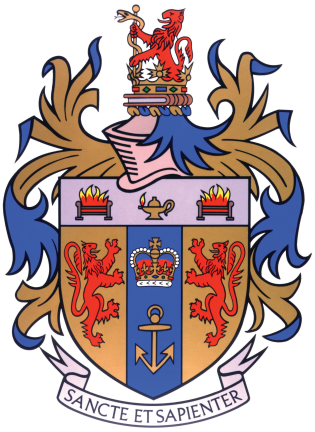
Savas, Okkes Emre

Awarding institution:
King's College London

King's College London

Faculty of Natural & Mathematical Sciences

Department of Informatics



Temporal-Numeric Planning with Control Parameters

Ökkeş Emre Savaş

A thesis submitted in partial fulfilment of the requirements of the degree of Doctor of Philosophy

First Supervisor: Professor Derek Long
Second Supervisor: Professor Maria Fox

Date of Submission: 16 April 2018
Date of Oral Examination: 22 June 2018
Date of Award: 1 November 2018

Keywords: temporal planning, numeric planning, domain-independent planning, PDDL, heuristic search, forward state space search, delete-relaxation heuristics, constrained resource planning, partially-ordered planning.

Abstract

Over the last decade, significant progress has been made in task planning to explore temporal and numerical features of the real world. Interaction between time and metric fluents and temporal coordination problems have been well-addressed by innovative approaches in planning and scheduling. A standardised language, PDDL, has enabled modelling the conceptual models and benchmarking the work of numerous researchers since it was released.

Although PDDL is an expressive modelling language, a significant limitation is imposed on the structure of actions: the parameters of actions are restricted to values from finite (in fact, explicitly enumerated) domains. There is one exception to this, introduced in PDDL2.1, which is that durative actions may have durations that are chosen (possibly subject to explicit constraints in the action models) by the planner. A motivation for this limitation is that it ensures that the set of grounded actions is finite and, ignoring duration, the branching factor of action choices at a state is therefore finite. Although the duration parameter can make this choice infinite, very few planners support this possibility, but restrict themselves to durative actions with fixed durations.

In this thesis we motivate a proposed extension to PDDL to allow actions with infinite domain parameters, which we call *control parameters*. We illustrate reasons for using this modelling feature and then describe a planning approach that can handle domain models that exploit it, implemented in a new planner, called POPCORN. We propose an extension to a delete-relaxation heuristic, called the Temporal Relaxed Planning Graph, to tackle problems with control parameters; especially in problems with producer-consumer relationship of actions. We show that this approach scales to solve interesting problems. We apply this work in a task and motion planning scenario to show that our work has a great potential of re-inventing the way task planners are used in robot navigation.

Acknowledgements

I would like to express my deepest gratitude and countless thanks to my supervisors (and my gurus in science), Maria Fox and Derek Long, for their invaluable input and persistent guidance on this bumpy journey. I feel privileged to be one of their few students.

I would like to thank Michael Cashmore and Bram Ridder for introducing me new hobbies, such as playing board games and bouldering, as well as having entertaining (and fruitful) brainstorming discussions.

I would like to thank the ones who collaborated with me Stefan Edelkamp, Daniele Magazzeni, Chiara Piacentini for their constant support and exceptional contribution to our work, especially Dan who has always been more than a colleague but also a good friend.

I would like to thank to my PhD examiners Nick Hawes and Rong Qu; my favourite authors in AI Planning Andrew Coles and Amanda Coles; true friends Salur Basbug, Elliott Fairweather and Fei Gao; my ICAPS mentor Chris Beck; and all of the members of the AI Planning group at King's College London for their support and guidance.

My PhD studies were funded by European Commision Seventh Framework Programme for Research and Technological Development (FP7) as a part of SQUIRREL project under grant agreement No 610532.

Dear Halil dede and Dear Tyson – you will always be remembered...

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Technological improvements have facilitated the automation of industrial processes and mechanical systems for decades. As the machines become increasingly capable, the science of applying natural intelligence to machines (also known as Artificial Intelligence (AI)) also becomes more applicable in real world practice. At the core of the problem of obtaining intelligent behaviour is the problem of choosing which activities to execute and when to execute them. Automated planning is a field of AI addressing these problems with reliable and reproducible solutions. It extends the capabilities of systems with *full autonomy* that is highly essential in most robotics applications. Two of the most outstanding automated planning achievements in history are the Remote Agent (Muscettola et al., 1998) and the ongoing Mars Exploration Rover (MER) missions of NASA. The Remote Agent is an autonomous self-repair software used in the *Deep Space 1* spacecraft developed by the Ames Research Center and the Jet Propulsion Laboratory at NASA. It was the first AI spacecraft control system that successfully demonstrated the ability to plan activities, diagnose and respond to malfunctions in the spacecraft without any human supervision. The MER missions of NASA involve two rover vehicles, Spirit and Opportunity, that have been exploring the planet Mars since 2003. In this mission, the objective is to collect rock and soil samples that can provide evidence to the past existence of water on planet Mars. The rovers use a framework, called MAPGEN, to build an activity plan to execute on each Martian day (Ai-Chang et al., 2004). Another AI planning achievement is the recent work of (Cashmore et al., 2014), which explores the problem of autonomous maintenance and inspection of a seabed facility. The process is carried out by autonomous underwater vehicles (AUVs) without any human intervention. Exploiting full autonomy is a requirement as human control is considerably hard and limited in these missions.

For more than a decade, automated planning approaches have been constantly evolved to capture complex attributes of real world applications. Temporal and numeric planning is a research area that is concerned with capturing metric and time-related attributes of the environment, such as instantaneous use of numeric resources, replenishment and consumption of resources in continuously evolving time (e.g. filling a fuel tank) and the requirement of simultaneous execution of activities. Many real world applications require planning and controlling of these attributes. For instance, in order to achieve full autonomy in an AUV, it needs to have the complete control over its time and numeric resources; such

as the velocity, the fuel level, light source settings and the duration of activities. This area has been identified as a challenge and has been tackled as a search problem by numerous researchers for more than a decade (Ghallab and Laruelle, 1994; Laborie and Ghallab, 1995; Smith and Weld, 1999; Hoffmann, 2003; Cushing et al., 2007; Do and Kambhampati, 2014; Eyerich et al., 2012; Piacentini et al., 2013; Bryce et al., 2015; Bajada, 2016). The main challenge in this field is the computational complexity of the search (Bylander, 1994; Rintanen et al., 2007; Aldinger et al., 2015), as time and numbers are dynamic features of the environment that can easily cause infinite decision points in search. Planning approaches rely on restricting these decision choices (i.e. corresponding to action choices) to finite values prior to planning, so that the choices on these attributes are always in finite ranges. The scope of this thesis is narrowed down to temporal and numeric planning assumptions and applications, in which these attributes are not necessarily drawn from finite number of choices, instead they are kept flexible (subject to an arbitrary collection of constraints), in order to reflect the true nature of these attributes (that are often kept flexible and managed efficiently in control systems).

Numerous temporal-numeric planners rely on forward state space heuristic search (Bonet and Geffner, 2001; Hoffmann, 2003; Helmert, 2006; Coles et al., 2009b; Fernández-González et al., 2015a), as it has shown effective performance solving complex planning problems. We employ the POPF planner (Coles et al., 2010) as our base system, which is a partially ordered temporal-numeric planner that exploits forward state space heuristic search. It reasons with the full semantics of the PDDL2.1 modelling language (Fox and Long, 2003), including continuous and discrete numeric change, and actions with flexible durations. It uses the Temporal RPG (Coles et al., 2009c, 2008a) heuristic, which derives and solves a simplified version of the problem for the purpose of guiding the search. The assumptions of the thesis are full observability, determinism, explicit continuous time and infinite state world (stated in detail in Section 2.1).

1.1 Motivation

The main motivation of the thesis was driven by problems that require the determination of numeric attributes at the end of the planning process, not by deciding from a set of options during search. We can describe these problems with a simple daily life example. Suppose that we are planning to go to a mall for shopping and we do not have enough cash in our pocket. The plan for this problem must include visiting a cashpoint to collect cash before going to the mall. However, we do not know precisely how much cash to withdraw when we are at the cashpoint. It can only be revealed once the entire plan of the day is constructed. The withdraw action should allow flexibility on this attribute (i.e. the amount of cash that can be withdrawn), so that the planner has freedom to determine the value once the rest of the plan is generated under an accumulated set of numeric constraints.

Another example to the necessity of flexible attributes is determining the locations that an agent must visit during navigation. In robot navigation, a task planner finds a meaningful symbolic plan while the motion planner finds a collision-free trajectory. In most task and motion planning integrations including the work of (Cambon et al., 2009; Erdem et al., 2011; Dornhege et al., 2009; Garrett et al., 2015; Cashmore et al., 2015), the task planners

model locations as discrete notion, where they can be obstructed by an unknown object (i.e. an object that the task planner does not initially know about) or they can be unreachable during execution (due to physical limitations), which can lead the task planner to re-plan. Modelling locations as continuous regions is possible using flexible attributes, which allows the task planner to reason about the estimated dynamics of the environment and provides a wider range of location coordinate options to the motion planner. We survey the possibility of this scenario as a case study in Chapter 5.

There are certainly other real world examples, in which some activities can desirably have multiple flexible attributes (or parameters) and they can be managed (or regulated) by the planner itself or by an external control system. We call these parameters *control parameters*, since they represent the control dynamics of the system in planning problem. Some of the real world examples that can benefit from modelling and reasoning with these parameters are:

- Production planning and inventory control: the order quantity, the lot size,
- Robot navigation: the torque of a motor, the velocity and acceleration,
- Refinery operations (e.g. thermal equilibrium, mixing liquids and chemical reactions): the quantity of a reactant, the heat transfer rates,
- Power management in unmanned vehicles (i.e. AUVs and UAVs): the battery charge rate per second
- Space applications (e.g. controlled spacecraft landing, orbiting objects): the thrust force of the vehicle.

1.2 Contributions of the Thesis

The main contributions of the thesis are:

- Extending the PDDL modelling language to declare and exploit typed numeric action parameters (i.e. control parameters), whose values are taken from infinite domains, in PDDL action schemas.
- Developing a forward state space heuristic search planner, called POPCORN, that can handle linear pre- and post-conditions and effects with multiple control parameters.
- Combining a continuous time relaxation, called infinity analysis, with an incremental graph expansion mechanism to avoid common relaxation issues in problems with repetitive resource transfer with control parameters. The resulting relaxation is called *refined infinity analysis*.
- Extending a delete-relaxation based temporal heuristic, the Temporal RPG, to include refined infinity analysis.
- Proposing an alternative modelling approach in a robot navigation scenario that describes locations as continuous regions rather than modelling as discrete objects.

1.3 Thesis Statement

This thesis explores the claim that:

> *A temporal-numeric task planner equipped with action-specific numeric parameters with large domains is capable of generating sophisticated and pragmatic numeric reasoning coupled with efficient search.*

1.4 Thesis Outline

The rest of the thesis is structured as follows.

In Chapter 2, we provide the details of the assumptions we used in the thesis. Then, we describe the most commonly used automated planning paradigms in an increasing order of complexity (i.e. classical, numeric, temporal planning paradigms), where we present the problem statement of each paradigm associated with their solution. We then introduce the PDDL modelling language (McDermott et al., 1998) that is the most well-known standardised planning language. Later, we describe the planning as a search problem identifying popular search and heuristic algorithms. We describe different heuristic evaluation techniques and their use in search. We conclude with brief descriptions of important planning systems, including POPF, and summarise their differences and clearly identify the position of our research among the state-of-the-art.

In Chapter 3, we propose an extension to the PDDL2.1 language, which enables encoding action parameters that are allowed to take their values from finite domains (i.e. control parameters). We describe the effects of control parameters in the search space and propose a way of handling them in forwards heuristic search framework, which makes use of a mathematical solver (i.e. LP) for tackling numeric constraints. We propose a minor extension to the Temporal RPG heuristic that enables reasoning with control parameters. We compare and contrast our approach with our closest competition, Scotty (Fernández-González et al., 2015a). We describe how we encode these parameters in the LP models. Then, we provide the formal definition and solution of planning with control parameters problem. We describe a way of managing flexible plans. We implement this work on the POPF planner, resulting in a new system called *POPCORN*. We conclude the chapter with an elaborate set of experiments with POPF and Scotty.

In Chapter 4, we investigate the weaknesses of the heuristic used in POPF, the Temporal RPG, in temporal-numeric problems with control parameters (specifically, producer-consumer problems). We identify the existence of control parameters in temporal setting significantly reduces the basic-informedness of the heuristic. We survey the reasons why existing heuristic approaches are inefficient in these problems and propose an extension to the Temporal RPG that considerably improves its informedness. We present a common production planning problem, in which numeric resources are repeatedly produced and consumed. We categorise certain actions as producers and consumers based on their pre- and post-conditions and effects in the domain model. Then, we formulate useful partially-ordered patterns to provide further informedness to the heuristic. We implement this heuristic on the POPCORN planner, resulting in a system called *POPCORN-R* (that is POPCORN with Refined Infinity Analysis) and benchmark it with POPCORN on a set of highly complex

producer-consumer problems and the extended version of Scotty on one of their example domains.

In Chapter 5, we present the use of control parameters in the task and motion planning (TAMP) integration as a case study. We present an alternative modelling approach to robot navigation, where the locations are modelled as continuous regions, not as location objects or discrete coordinates. We investigate the state-of-the-art modelling techniques and their weaknesses in robot navigation. We also survey the most well-known TAMP approaches. We present the technique in a scenario used in the SQUIRREL EU robotics project. We investigate the modelling technique, plan generation and execution in depth. We evaluate the technique using POPCORN to solve models with continuous regions, POPF and TFD (Eyerich et al., 2012) to solve models with discrete locations. We present the evaluation results for 1600 problem instances run on 3 planners.

Chapter 2

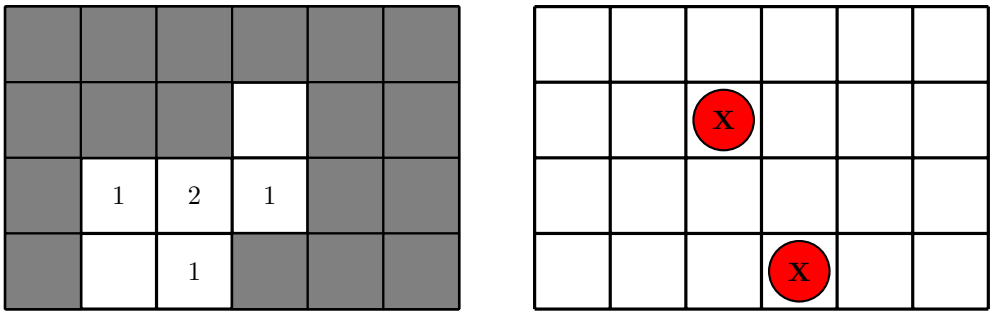
Background

In this chapter, we provide descriptions of previous work that are relevant to this thesis. The chapter is structured as follows. We start with a brief introduction to the automated planning, then provide detailed descriptions of the main components of a domain-independent planning problem describing suitable languages, models and algorithms; as well as systems that implement these components. We conclude the chapter with a concise survey on the integration of task and motion planning, as we conduct a case study on this subject in this thesis.

2.1 Introduction to Automated Planning

An important problem of autonomous behaviour is the problem of choosing which activities to execute next (Geffner and Bonet, 2013). There are various ways to address this problem. In the programming-based approach, the programmer explicitly specifies which action to execute in certain circumstances defined in a program. In the learning-based approach, the programmer does not impose any decision to the agent, but the decision relies on experience learned from the environment. In the model-based approach, the decision is neither dictated by the programmer nor learned from the environment but it is derived from a model of possible activities, the world state and the goals. Each of these approaches have different limitations. In programming-based approach, the programmer is responsible for foreseeing all possible contingencies of the problem and usually results in implementing a fragile system. The learning-based approach is limited to the experience the agent possess. The model-based approach takes a list of actions, goals and the world state as the input and considers all possible scenarios, but it is limited to the computational complexity of the problem.

The Minesweeper game is a single-player video game, where the objective is to clear a rectangular board containing bombs without detonating any of them, with the help of numeric clues indicating the number of bombs in the neighbouring tiles. The three approaches we discussed earlier would offer solutions to this problem as follows. The programming-based approach requires the actions to be taken hard-coded in a program. The learning-based approach requires learning process in a set of training runs in the Minesweeper game environment. The model-based approach can derive a solution solely from the world



Unsolved Minesweeper Solved Minesweeper

Figure 2.1: The illustration of the partially solved (on the left) and the solved (on the right) Minesweeper game.

state, actions list and the goal conditions.

Automated planning is a model-based approach to autonomous behaviour where the agent chooses the next action from the actions list considering the world state and the goals. The authors of The Automated Planning textbook (Geffner and Bonet, 2013) splits model-based approaches to action selection problem into three parts: the *models* that express the dynamics, feedback, and the goals; the *languages* that express these models in compact (and standardised) form; the *algorithms* that exploits the representation of models for generating the behaviour.

Figure 2.1 illustrates a small 4×6 unsolved Minesweeper game with two bombs hidden. In this figure, the bottom left tile is labelled with the location (0,0) and the coordinates of each tile is incremented by 1 on right and upper directions. In this problem, we have 24 boolean variables that control whether each corresponding tile is already discovered or not. Another 24 boolean variables control whether each tile has a bomb or not. A *state* for this problem is a valuation over these variables. The figure on the left is the *initial state* and the one on the right is one of the possible solutions to the problem (one of *goal states*). At the initial state, there are 6 already discovered tiles. There are $2^{18} = 262144$ possible states (including possible goal states) in total from the initial state. The available action to the agent is to *unveil a tile*. The *event* of denotation of any bomb results in failure of the game. Initially, we do not know where the bombs are hidden and what numeric clue we will obtain when revealing a tile, so the state is *partially observable*. The system obtains knowledge about the hidden variables using the observable variables. For instance, when we consider the clues on tiles (3,1) and (4,2), one can easily recognise that the tile (4,1) must be a bomb. As the tile (4,3) indicates that there are no surrounding bombs to this tile, the tiles (3,3), (3,4) and (4,4) can be marked as safe without any hesitation. Following this move, one can realise that there must be a bomb in tile (2,3), which results in solving the puzzle successfully.

It is possible to solve this problem instance by writing a hard-coded program where it takes the initial and goal states. As a result, the program can perform a behaviour that is somewhat similar to (or indistinguishable from) the behaviour of a human. This program is likely to fail solving other problems, even if they follow similar set of logical rules. However, in automated planning, the objective is to design a program that is not only limited to

the Minesweeper game. We expect that it can solve *any* problem that follows a certain mathematical structure. The mathematical structure of the problem becomes apparent in its planning model (discussed later in this chapter). There are various examples of problems that have similar mathematical (or logical) representations to the Minesweeper game, such as the Wumpus (Russell and Norvig, 2010) and the Solitaire games. The ability of identifying the characteristics of a problem based on their representations and solving identical ones using generic algorithms constructs the core of the automated planning.

The environments of Minesweeper-like games are partially observable (i.e. the agent holds partial knowledge about the environment) and non-deterministic (that is when the execution of an action may or may not generate a new state). As might be expected, not every planning problem is similar to the Minesweeper game. Various games are deterministic, as in the Rubik's cube, where the effects of an action always yield to a new child state. Some restrictive assumptions made on the representation of the environment clarify the scope of the planning model (Ghallab et al., 2004). The assumptions are mainly based on the characteristics of the actions and the states of the environment; and they can be enlisted as follows:

- Deterministic vs. non-deterministic: the planning problem is deterministic if the execution of the same action at the same state always yields to the same child state.
- Finite vs. infinite state world: in case the problem yields to a finite set of states, it is considered to be a finite state world. It is worth mentioning that an infinite state world could arise in two ways: arithmetic operations on numeric variables can yield to infinitely many child states (an example problem is provided in Section 2.2.2) and/or the domain of reachable values for one or more numeric variables could be infinite (e.g. $v :: [1, \infty]$, where v is a numeric variable).
- Full vs. partial observability: the state of the environment is fully observable if the agent holds the complete knowledge of the system.
- Implicit time: the actions do not have any duration so that the state transition is instantaneous.

Different planning paradigms arise depending on which of these restrictive assumptions are relaxed in the environment. In this thesis, we study automated planning in deterministic, fully observable, explicit time and infinite state world environment. This implies that:

- Infinite state world: we tackle problems in which the numerical action parameters takes their values from infinite domains that create infinite set of states. We give the details of this restriction in Section 3.
- Determinism: the execution of an action takes the system to a single other state.
- Full observability: one has the complete knowledge about the state of the system.
- Explicit continuous time: in planning domains, we consider action duration and concurrency. The time is also taken into account during transition from one state to another and it can be a continuous property of the problem. The details of this assumption is considered in Section 2.2.3.

The formulation of mathematical structures and the detection of the causal relations of problems is the main concern of automated planning. In the next section, we present the general formalisation of different planning problems in the increasing order of complexity. We start with the classical planning problem and conclude the section with the temporal-numeric planning problem formalism. In this thesis, we are particularly interested in temporal-numeric planning problem as it allows an explicit representation of time and numeric resources. In further sections, we survey *a standardised modelling language*, PDDL (McDermott et al., 1998; Fox and Long, 2003; Hoffmann and Edelkamp, 2005; Fox and Long, 2006; Gerevini et al., 2009), adopting these formalisms, and conclude the chapter with a survey of the highly-reputed *algorithms* that are used to find a solution for the problem.

2.2 Representations of Planning Problems

Automated planning is specialised into different paradigms depending on the assumptions that are taken in the environment and the type of transitions considered between states. First, we survey classical planning as it is considered to be the starting point of AI planning research. Second, we study numeric and temporal planning paradigms, which introduce numeric and temporal features of the world into the planning problem.

2.2.1 Classical Planning

Classical planning is a deterministic, fully observable, discrete and static planning paradigm (Russell and Norvig, 2010). It is concerned with finding a sequence of actions that takes the system from an initial state to a goal state, where the effects of actions are deterministic and known. Classical planning can be considered as a path finding problem over a directed graph whose nodes represent the states of the system and edges represent state transitions of applying actions. Geffner and Bonet formalise the classical planning model describing the relevant concepts as follows (Geffner and Bonet, 2013).

Definition 1 (Classical Planning Model) *The classical planning model is defined as a tuple $\langle S, A, s_0, G, T\rangle$, where:*

- *S is a finite set of states, where each state consists of a set of grounded literals,*
- *A is a finite set of grounded actions. $A(s) \subseteq A$ represents the set of actions in A that are applicable in state $s \in S$,*
- *$s_0 \in S$ is the initial state. It consists of a finite set of literals that initially hold true,*
- *$G \subseteq S$ is the non-empty set of goal states,*
- *$T(a, s)$ is the state transition function that drives the state s to s' with execution of action $a \in A(s)$. s' can be represented as $s' = T(a, s)$.*

In this definition, *a grounded literal* refers to a first-order literal of the form $p(q_1, \ldots, q_n)$, where p is a predicate of arity n with arguments q_n. Each of its arguments is bounded to an object specified in the problem description. For instance, `at(?t ?loc)` is a predicate and

`at(truck1 location1)` is a grounded literal. *A grounded action* (or instantiated action) refers to an action schema, whose arguments are bounded to objects defined in the problem.

There are two general formalisations of the classical planning model: the propositional logic formalisation and the multi-valued variable formalisation (based on SAS^+) (Bäckström and Nebel, 1995; Helmert, 2006). We only survey formalisation based on propositional logic as it is the relevant representation to the thesis. In this formalisation, predicates describe the static or non-static features of objects (i.e. the agent can traverse between two specified location objects in a navigation problem), and *boolean propositions* are added or deleted (to/from the state world) when an action is applied. The model can be shown as follows:

Definition 2 (Propositional Formalisation of Classical Planning Problem) *The classical planning problem (or a task) formalised using propositional logic is a tuple of* $\langle F, A, I, G\rangle$*, where:*

- F *is a finite set of grounded literals (or facts),*
- A *is a finite set of grounded actions derived from action schemas specified in the domain,*
- $I \subseteq F$ *is the finite set of grounded literals that hold true at the initial state,*
- $G \subseteq F$ *is the finite set of grounded literals that needs to be achieved in order to reach the goal state.*

In the definition, each action $a \in A$ is denoted as a tuple $a = \langle pre,\ eff^+,\ eff^-\rangle$, where $pre(a) \subseteq F$ denotes the preconditions, $eff^+(a) \subseteq F$ denotes add effects, $eff^-(a) \subseteq F$ denotes delete effects of action a. The effects of a can be shown as $eff(a) = eff^- \cup eff^+$. An action a is applicable at state s iff $pre(a) \subseteq s$. The result of applying action a at state s is a new state $s' = s \setminus eff^- \cup eff^+$.

The propositional formalisation of a classical planning task is commonly known as the STRIPS formalism (Fikes and Nilsson, 1971). The STRIPS formalism is based on two important assumptions. First, it assumes all facts that are not known to be true, are false (*closed world assumption* (Reiter, 1977)). Second, it assumes that only actions can update the world state with their own effects (*STRIPS assumption*). In other words, it assumes that applying an action updates the values of literals that are affected by the effects of this action and the values of the rest of the literals remain unchanged. The state transition function, $T(a, s)$, in STRIPS assumption can be derived from the following equation:

$$T(a,s) = \begin{cases} s \setminus eff^-(a) \cup eff^+(a), & \text{if } pre(a) \subseteq s \\ undefined & otherwise \end{cases}$$

The solution to the classical planning problem can be defined as:

Definition 3 (Classical Planning Solution) *Suppose that the* π *is a sequence of applicable actions* $\pi = \{a_0, a_1, \ldots, a_n\}$ *that generates a sequence of states* $\{s_1, \ldots, s_{n+1}\}$*.* π *is a solution to the classical planning problem if* $pre(a_i) \subseteq s_{i+1}$ *and* $s_{n+1} \in G$*, where* $i = \{0, \ldots, n\}$*.*

The cost of applying each action is constant and it is equal to 1. Thus, the cost of the plan is equal to the plan length. The shortest plan yields to *(cost-)optimal plan*, which refers to (in sequential classical planning) a plan that contains fewer actions than other plans for the problem. The systems that find optimal plans are optimal planners. Satisficing planners are the ones that generate good plans (or *satisficing plans*) which are not necessarily optimal.

2.2.2 Numeric Planning

Numeric planning introduces numeric variables to planning that extends the planning task beyond the propositional logic formalisation. The numeric variables can take their values from finite (or potentially infinite) domains, such as integer, real numbers or rational sets. If the numeric domain is finite, the task can be transformed into a planning problem over a finite set of multi-valued or boolean variables. The existence of numeric variables enables defining goals and preconditions as multi-valued functions, such as $f(x) \geq c$ or $f(x) > c$; where the variable c denotes a constant number and $f(x)$ is a function over numeric variable x. This contrasts to classical planning where the preconditions and goals are boolean atoms (that can either be true or false). In real world applications of numeric planning, the numeric variables (also known as state variables) include dynamic-changing resources of the environment, such as the fuel level and the velocity of a vehicle, the volume setting. It is worth noting that the introduction of numeric variables makes the planning problem *undecidable* (Helmert, 2002), as the search space can become infinite and the search algorithms may run forever without returning a solution or determining that there is a solution. We give an example below, for which the search algorithms will have to run forever and would not return a solution.

Numeric planning task can be easily mapped into a classical planning task where the goal state is reached from an initial state by applying a sequence of deterministic actions. The only difference between the two is that the numeric planning can yield to infinite number of states. This problem arises even if the numeric variables are restricted to take their values from finite domains. Geffner and Bonet describe this problem using the following numeric planning example (Geffner and Bonet, 2013). Consider a planning problem where an action assigns the value of variable x to its half (i.e. the assignment effect is $x := x/2$). The initial value of x is 1 ($x = 1$), and the target value is 0 ($x = 0$). This problem is unsolvable using search algorithms for finite plans. This problem and many more challenging numeric planning problems have been addressed over the last decade and can be found in (van den Briel et al., 2007; Löhr et al., 2013; van den Briel and Kambhampati, 2005; Gerevini et al., 2008).

The formulation, search and the use of heuristics has become the major challenges in numeric planning for more than a decade. The 3rd International Planning Competition (IPC-3) (Long and Fox, 2003) can be considered as the turning point in this direction. A standardised planning language (PDDL2.1 (Fox and Long, 2003)) is introduced for the use of the competition, which enabled other researchers to benchmark their approaches. Since then, various search and heuristic algorithms have been proposed to find solutions for finite plans (if they exist) (Hoffmann, 2003; Coles et al., 2008b; Piotrowski et al., 2016; Scala et al., 2016b; Francès and Geffner, 2015).

As a step towards formalising the numeric planning problem, the classical planning model and its formalisation given in the Definition 1 and 2 are extended as follows. In addition to a set of literals (F), a set of numeric variables ($\mathbf{v}$) is defined. The size of $\mathbf{v}$ is r that can be represented as $\mathbf{v} = \{v_1, \ldots, v_r\}$. Suppose that $F(s) \subseteq F$ and $\mathbf{v}(s) = \{v_1(s), \ldots, v_r(s)\}$ denotes a set of literals and a vector of rational numbers (the values of each variable), respectively, at state $s \in S$. The state notation in Definition 1 is extended such that the state s is a pair $s = \langle F(s), \mathbf{v}(s)\rangle$.

The numeric planning problem is an extension of the classical planning problem in propositional logic given in Definition 2 and it is denoted as a tuple of $\langle \mathbf{v}, F, A, I, G\rangle$. Each action $a \in A$ is denoted as a tuple $a = \langle pre,\ pre^n,\ eff^+,\ eff^-,\ eff^n\rangle$, where $pre(a) \subseteq F$ denotes a set of propositional preconditions, $pre^n(a)$ denotes a set of *numeric preconditions* over the state variables $\mathbf{v}$, $eff^+ \subseteq F$ denotes propositional add effects, $eff^- \subseteq F$ denotes propositional delete effects, $eff^n(a)$ denotes a set of *numeric effects* over the state variables $\mathbf{v}$ of action a.

Let exp and exp' denote arithmetic expressions over $\mathbf{v}$ and the rational numbers that are formulated using the operators $\{+, -, *, /\}$, $comp$ denotes comparison operators $\{\geq, >, =, <, \leq\}$, op denotes assignment operators {+=, -=, :=, *=, /=}. A numeric precondition and a numeric effect are defined as:

Definition 4 (Numeric Precondition) *A numeric precondition is a tuple* $\langle exp, comp, exp'\rangle \in pre^n(a)$.

Definition 5 (Numeric Effect) *A numeric effect is a tuple* $\langle v_i, op, exp\rangle$, *where* v_i *is a state variable in* $\mathbf{v}$.

An action a is applicable at a state s if and only if all preconditions pre(a)$\subseteq s$ and all numeric preconditions in $\text{pre}^n(a)$ hold in s. The result of applying action a at s is a new state $s' = s \cup eff^+ \cup \mathbf{v}(s') \setminus eff^-$, where $\mathbf{v}(s')$ is the value vector that results from applying $eff^n(a)$ over numeric variables in $\mathbf{v}$.

2.2.3 Temporal Planning

Temporal planning refers to planning using actions with durations allowing them to be executed concurrently in certain conditions. The earliest instance of reasoning with time in planning is the introduction of *timelines* and *chronicles* used in IxTeT (Ghallab and Laruelle, 1994). The chronicles consist of temporal constraints over a set of state variables and a timeline is a chronicle of a single state variable. Their work has pioneered the most important temporal planning tools, such as simple temporal network (STN).

The temporal-numeric planning problem is an extension of the numeric planning problem in propositional logic where it is denoted as a tuple of $\langle \mathbf{v}, F, A, I, G\rangle$. The representation of actions in A is extended, so that they include the duration of actions. The resulting actions are called *durative actions*, and can be defined as follows.

Definition 6 (Durative Action) *A durative action* $a \in A$ *is a tuple* $a = \langle dur,\ pre_{\vdash},\ eff_{\vdash},\ pre_{\leftrightarrow},\ pre_{\dashv},\ eff_{\dashv}\rangle$, *where:*

- *$pre_{\vdash}$ and $pre_{\dashv}$ denotes the start and the end conditions of the action a, respectively, where each consists of propositional (pre) and numeric (pre^n) conditions that must hold.*
- *$pre_{\leftrightarrow}$ denotes the propositional and numeric invariant conditions of the action a that must hold between the start and the end of the action.*
- *$eff_{\vdash}$ and $eff_{\dashv}$ denotes the start and the end effects of the action a. Suppose that eff_x denotes each given collection of effects where $x \in \{\vdash, \dashv\}$. Each eff_x consists of:*
  - *eff_x^+ denotes literals to be added to the world state,*
  - *eff_x^- denotes literals to be deleted from the world state,*
  - *eff_x^n denotes the numeric effects acting on state variables.*
- *dur denotes the duration constraints of the action a that constrains the time length between start and end of the action a. Each duration constraint consists of a special numeric parameter, `?duration`, that represents the duration of the action.*

The duration parameter `?duration` can appear in the numeric effects of the action (eff_x^n). This addition changes the arithmetic expressions used in numeric effects. Suppose that exp_e denote an arithmetic expression over $\mathbf{v}$, `?duration` parameter and rational numbers, *op* denotes assignment operators {+=, -=, :=, *=, /=}. A numeric effect becomes a tuple $\langle v_i, op, exp_e \rangle$, where v_i is a state variable in $\mathbf{v}$. It is also worth noting that the duration parameter does not appear in numeric conditions. Each durative action creates two time-points: at the start and at the end of the action.

The plan (or a solution) to the temporal-numeric planning problem can be defined as:

Definition 7 (Temporal-Numeric Plan (π)) *is a list of timestamped actions with duration in π. It is shown as:*

$$\pi = \{\langle a_i, ts_i, \Delta t_i \rangle \mid \forall i = \{0, \ldots, n\}\},$$

where n is the plan length, $ts_i \in \mathbb{R}$ is a timestamp (the time at which the action a_i is applied), $\Delta t_i \in \mathbb{R}$ is the duration of the action a_i. The plan generates a sequence of at most $2 \cdot n$ states (each is denoted by $s(t)$) by applying the start and the end of action a_i at time-points ($t = ts_i$) and ($t = ts_i + \Delta t_i$), respectively. Additionally, the following conditions must hold true:

- *the duration constraints, dur, of each action in π are satisfied,*
- *$pre_{\vdash}(a_i)$ and $pre_{\dashv}(a_i)$ hold in states $s(ts_i)$ and $s(ts_i + \Delta t_i)$, respectively, such that $\forall i = \{0, \ldots, n\}$,*
- *$pre_{\leftrightarrow}(a_i)$ hold in every state $s(t)$, where $ts_i < t < ts_i + \Delta t_i$, such that $\forall i = \{0, \ldots, n\}$,*
- *The trajectory of states generated by application of the actions yields a final state that satisfies the goal.*

2.2.3.1 Temporal Coordination

The introduction of time in planning has enabled researchers to apply planning technology in more realistic problems than the sequential puzzle problems (e.g. Minesweeper game). One of the most important challenges posed in temporal planning is the ability of executing multiple actions in parallel, where the non-temporal planning approaches are inadequate. In classical planning actions appear in a sequence where the concurrency between actions is abstracted.

Although PDDL2.1 introduces the expressive model of durative actions, earlier temporal planners, such as MIPS-XXL (Edelkamp and Jabbar, 2006) and LPG (Gerevini et al., 2006), translate PDDL temporal actions into a single temporally-extended STRIPS action. This semantic representation is called as *action compression* and it is introduced in TGP (Smith and Weld, 1999). However, applying action compression to durative actions can cause incompleteness (Coles et al., 2009c). Later, the idea of *required concurrency* is introduced by Cushing et al. (Cushing et al., 2007) to characterise the cause of incompleteness in some temporal problems.

2.2.3.2 Interaction between Time and Numeric Resources

The temporal-numeric planning formalism models problems with temporal coordination and *discrete numeric change*. The numeric effects update state variables instantaneously: either at the start or at the end of the action. Many real world applications require us to initiate and control continuous processes, in which the numeric variables are updated in time-dependent ways. For instance, dropping an object from a height initiates a continuous process of *falling* where the height is continuously updated over time. In this example, the world state is updated with a discrete change at a step (e.g. dropping the object) and a continuous change between steps (e.g. falling).

Any planning problem requiring interaction between numeric and time properties (as in dropping an object example) are called *hybrid systems* and they can include *mixed discrete-continuous* numeric change in the world. The theory of *hybrid automata* provides an underlying conceptual formalism for modelling and reasoning about mixed discrete-continuous systems (Henzinger, 2000). Many researchers have proposed heuristic and search algorithms addressing this problem (Coles et al., 2009b; Li and Williams, 2008; Penna et al., 2009; Bryce et al., 2015; Bajada et al., 2015).

2.3 Modelling Language

Having presented the temporal-numeric planning model, we report the PDDL language that expresses the model in a standardised way.

2.3.1 The PDDL Language

The Planning Domain Definition Language (PDDL) (McDermott et al., 1998) was introduced to run The First International Planning Competition (IPC) in 1998 (McDermott, 2000). The objective of the competition is to benchmark state-of-the-art planning systems.

The first version of PDDL was inspired by STRIPS and ADL (Pednault, 1989) languages. It includes typed object parameters in addition to the STRIPS and ADL features.

In PDDL, the planning task is modelled in two parts. *The domain model* includes the action schemas and their typed constant objects, predicates and model requirements. *The problem instance* includes enumerated objects and their types, the facts that are true at the initial state and the facts that must hold true to reach the goal conditions.

Since IPC-1, PDDL has evolved with each competition, resulting in various official versions and extensions available today. These languages are described in (Fox and Long, 2003; Hoffmann and Edelkamp, 2005; Gerevini et al., 2009; Kovacs, 2011; Younes and Littman, 2004; Sanner, 2010). Various other PDDL extensions have been proposed, but they are not used as official IPC languages, and they can be found in (Dornhege et al., 2009; Fox and Long, 2006; Kovacs, 2012). PDDL+ (Fox and Long, 2006) is one of these extensions that allows modelling mixed discrete-continuous planning problems through *exogenous events* and *continuous processes*. In this thesis, we use the PDDL2.1 (Fox and Long, 2003) extension as our base language, since it is the most frequently used PDDL version describing temporal-numeric problems. It extends Lifschitz' STRIPS semantics (Lifschitz, 1987) to encode numeric and conditional effects, as well as temporally designed actions.

2.3.1.1 The PDDL2.1 Language

PDDL2.1 introduces four levels of expressiveness. Each level introduces an additional expressive power to the previous one and they are summarised as follows:

- *Level 1:* STRIPS Encoding,
- *Level 2:* Numeric extensions,
- *Level 3:* Discretised durative actions,
- *Level 4:* Continuous durative actions,

All levels (except level 1) use the arithmetic operators to construct numeric expressions (as in the form described in Section 2.2.2). The numeric preconditions are constructed using these numeric expressions and comparison operators. The numeric effects use numeric expressions and definitive syntax for assignment operators (`increase`, `decrease` and `assign`). Figure 2.2 captions the pouring water between jugs example presented by Fox and Long (Fox and Long, 2003). The `(amount ?j)` and `(capacity ?j)` are *typed* state variables. The precondition of `pour` action is a numeric constraint over these variables that can be formulated as `(capacity ?jug2) - (amount ?jug2) >= (amount ?jug1)`. The first numeric effect of the action assigns the value of `(assign ?jug1)` to zero, whereas the second one increases the value of `(amount ?jug1)` by the value of `(amount ?jug2)`.

In PDDL2.1 language, the conditions and effects in a durative action must be *temporally annotated*. The annotation of a condition specifies whether it must hold *at the start* of the action (the time point at which the action is applied), *the end* (the point at which its end appears) or *over the interval from the start to end* (invariant over the action duration). The annotation of an effect specifies whether the effect becomes true at the start or at the end of the action.

```
(define (domain jug-pouring)
    (:requirements :typing :fluents)
    (:types jug)
    (:functions (amount ?j - jug) (capacity ?j - jug))

(:action pour
 :parameters (?jug1 ?jug2 - jug)
 :precondition (>= (- (capacity ?jug2) (amount ?jug2)) (amount ?jug1))
 :effect (and  (assign (amount ?jug1) 0)
    (increase (amount ?jug2) (amount ?jug1))))
```

Figure 2.2: Pouring water between jugs example as described by Fox and Long (Fox and Long, 2003).

```
(:durative-action scan_ticket
 :parameters (?g - gate ?p - person)
 :duration (and (>= ?duration (readiness ?p)) (< ?duration 10))
 :condition (and (at start (near ?g)))
 :effect (and (at start (open ?g))
    (at end (not (open ?g)))))

(:durative-action pass_through_door
 :parameters (?p - person)
 :duration (= ?duration (walking_speed ?p))
 :condition(and
 (at start (open ?g))
 (over all (open ?g)))
 :effect (and (at end (passed))))
```

Figure 2.3: Going through the ticket gate example encoded in PDDL2.1 level 3

Figure 2.3 shows the main actions of an example, in which the objective is to pass through a ticket gate. In this example the execution of `scan_ticket` action creates an opportunity for at most ten time units allowing the execution of `pass_through_door` action. Observe that `pass_through_door` action uses fixed duration specification whereas `scan_ticket` uses flexible duration limits. In the fixed duration case, the duration of the action is determined by the value of the variable `(walking_speed ?p)` in the state. In the flexible duration case, the duration of the action is constrained in an interval. This case gives the planner freedom to choose the duration of the action in this interval at the time of execution.Having flexible duration creates a more complex branching choice in the search space than in the fixed duration case.

A continuous durative action includes numeric effects that update the state variable continuously over time. While the discrete effects occur either at the end or at the start of the action, the continuous effects occur *during the execution* of the action. The language allows increase/decrease effects on variables according to a fixed rate of change over time. The language includes a reserved syntax, `#t`, denoting the continuously changing time from the start of a durative action during its execution. The scope of `#t` is limited to each durative action. To illustrate the use of `#t`, consider the fuel consumption of a car, which

continuously decreases over time. The continuous decrease effect can be shown as:

```
(decrease (fuel_level) (* #t (consumption_rate)))
```

Plan Metrics: An important extension of PDDL2.1 in optimal numeric planning is *the plan metrics* field that can be optionally supplied in the problem specification. It specifies the basis on which the planner evaluates the problem. The numeric expressions that are desired to be optimised (i.e. *maximized* or *minimized*) are defined in the plan metrics field. An abstract example to the plan metric can be given as follows. Assume the modeller aims to minimise the linear expression $3x + y$ in planning procedure, where x and y are state variables. Then the metric plan can be shown as:

```
(:metric minimize (+ (* (x) 3) (y)))
```

2.4 Planning as Searching through Spaces

The planning problem can be considered as a path finding problem over a directed graph. It can be solved using different space search approaches. The first approach is based on searching through the world states, in which the nodes represent possible world states and the edges (or state transitions) represent action execution. This approach is called *state space search*. The second approach, called *plan space search*, is based on searching through plan spaces (Sacerdoti, 1975; Weld, 1994). In this search, the nodes represent partially specified plans and the edges represent *plan refinement operations*, which refers to the modifications to the partial plan (such as appending actions to the end of the partial plan).

There are three generic branching schema approaches to search through these spaces. The first approach, called *forward search* (or progression), starts the search from the initial state and progresses through states until a goal state is reached. The second approach, called *backward search* (or regression), starts the search from the goal and regresses back through states until the initial conditions are reached. The third approach, called *bidirectional search*, the backward and forward search algorithms are run simultaneously and the solution is found once the algorithms meet at a common state.

Partial order planning refers to the formulation of the planning problem as a search problem in partially specified plan space. The initial state represents an empty plan and the goal state represents a complete plan. The state transitions are the plan refinement operations. A partial plan is a tuple $\mathcal{PP} = \langle A, L, O, B \rangle$, where A is a set of actions, L a set of causal links, O a set of ordering constraints, B a set of binding constraints. Ordering constraints define a partial order on actions and binding constraints define a partial order on action parameters in $\mathcal{PP}$. B is an empty set if the actions are grounded. A solution is reached if each precondition (open condition) of actions is linked to an effect of another action and each unsafe causal link between actions is resolved. Plan space search uses the *least commitment principle*, which refers to adding an ordering constraint between actions to the partial plan only if it is strictly needed to reach the goal.

In this thesis, we focus on forward state space search planning. The system we use, POPF (Coles et al., 2010), makes use of partial ordering of actions during the search to improve its performance. In the next section, we mention the most well-known search algorithms and describe the one used in this system.

2.4.1 Search Algorithms

Search algorithms are split into two categories: *blind search* and *informed search*. Blind search algorithms are the ones that do not have any information about the search space (except the initial and the goal states). The most commonly used ones are *depth-first* and *breadth-first* search algorithms. Informed search algorithms obtain additional information about the problem in order to promote some states over others. These algorithms use a *heuristic*, which is an estimation of the cost to reach a goal. For this reason, informed search is usually called heuristic search. The most well-known informed search algorithms are A^* (Hart et al., 1968), *weighted* A^* (Pohl, 1970) and *greedy best-first search* (Pearl, 1985) algorithms. These algorithms are *complete*, where it is guaranteed to find a solution if there exists one. *Best-first search* is a generic way of referring to the class of complete informed search algorithms.

Note that not all informed search algorithms are complete. Hill climbing and enforced hill climbing (EHC) (Hoffmann and Nebel, 2001) are *local search* algorithms that can fail to find a solution. Even if it finds one, the solution is likely to be a *local optima* or a *plateau*.

Using blind search (and even complete search) algorithms in planning problems (especially in numeric planning) is non-viable, as they can severely suffer from the size of the search space. Many planning systems make use of local search to find a solution with the least state evaluations (Bonet and Geffner, 1999; Hoffmann, 2003; Coles et al., 2009c).

Next, we give details of how heuristics are used in most informed search algorithms and concentrate on describing weighted A^* and enforced hill climbing informed search algorithms as they are used in our base system, POPF.

2.4.1.1 Informed (or Heuristic) Search

The heuristic evaluation methods (we will present them in Section 2.5) return a heuristic value, $h(s)$, which provides a quick-and-dirty estimate of the minimal cost to reach the goal from the state s (Geffner and Bonet, 2013). This value is used in *evaluation function*, $f(s)$, which incorporates heuristic information in search and it varies between informed search algorithms. We can describe the weighted A^* and enforced hill climbing algorithms as follows.

- *Weighted* A^* is a best-first search algorithm. It is a variation of A^*, in which the evaluation function is a weighted sum of the estimate cost of achieving the goal from a state s (i.e. $h(s)$) and *the accumulated cost* of achieving the state s (i.e. $g(s)$). Let w_g and w_h be real-valued weights, where $w_g, w_h > 1$. The evaluation function is defined as follows:
$$f(s) = w_g \cdot g(s) + w_h \cdot h(s) \tag{2.1}$$

- *Enforced Hill Climbing* is a local informed search algorithm, which is a variant of the hill climbing algorithm. It was introduced by Hoffmann and Nebel (Hoffmann and Nebel, 2001). Starting from a state s, it searches for the successor state s' of s (using the complete breadth first search algorithm) that is closest to the goal according to the relaxed heuristic distance measurement and keeps the states in a queue. When the breadth first search finds the closest better successor, where $h(s') < h(s)$, the queue is

deleted and the breadth first re-starts from the state $s = s'$. The search stops when it reaches a goal (a state, where $f(s) = h(s) = 0$) or it cannot find a better successor.

2.4.1.1.1 Helpful Actions The idea of automatically deriving heuristic information from the problem is not only limited to obtaining a heuristic value at a state. The idea is applied to derive structural information (of the search) in the form of *helpful actions* (or *preferred operators* (Helmert, 2006)) and it was first studied by (Hoffmann and Nebel, 2001). A set of helpful actions are a subset of applicable actions at a state s, which are more promising than the other applicable actions at s towards meeting the top-level goals of the problem (the goals that are not true in s). EHC algorithm ignores all the actions that are not helpful in a given state to enhance the search performance. Consequently, the branching factor of the problem is significantly reduced and this is how the EHC algorithm becomes incomplete. If the helpful action list is empty, the EHC immediately fails. However, the use of helpful actions is not limited to local search. A novel best-first search algorithm introduced by (Helmert, 2006) incorporates helpful actions. The algorithm uses two open-lists (one for helpful actions and one for all applicable actions) and alternates between them based on heuristic value.

The heuristic technique used in POPF makes use of helpful actions. We will give the details of its base heuristic and the process of extracting helpful actions in Section 2.5.1.2.

Having described the search algorithms, we categorise and briefly describe the most well-known heuristic evaluation methods used in planning problems (in an increasing complexity order starting from classical planning).

2.5 Heuristic Evaluations

The heuristic values are automatically derived from a problem. They are used to guide the search towards a goal. A useful heuristic is the one that is computationally fast and provides a good estimate of the cost to the goal (i.e. $h(s)$). Different heuristics are used depending on the problem we have (Edelkamp and Schroedl, 2012). For instance, the Euclidean distance is a good heuristic for path finding problems, whereas the sum of the Manhattan distances is a good heuristic for the sliding puzzle problem. In general, the idea of developing heuristics is based on reducing the complexity of the original problem to a simplified version (called *relaxation*). If the solutions to the original problem are also solutions to the relaxed version of the problem, the heuristic from the relaxation is indeed *admissible*. This is a preferred characteristic in a heuristic, but it is not a compulsory feature.

In general, the heuristic evaluation methods in domain-independent planning are divided into four categories (Geffner and Bonet, 2013):

– *Delete-relaxation* (Hoffmann and Nebel, 2001) is the most common domain-independent planning heuristic. It maps the planning task $\langle F, A, I, G\rangle$ into the relaxed version $\langle F, A^+, I, G\rangle$, in which the grounded actions in A has an empty delete effects list (i.e. ignores the effects of actions that remove predicates from the world state). In this relaxation, new facts are added by add effects while the delete effects are ignored. For instance, when a truck moves from a location to another it remains to be at both

locations at the end. Although the resulting plan, the relaxed plan, obtained from the planning task is meaningless and does not reflect the actual model of the world, it provides quick and efficient guidance in the search space.

– *Pattern databases* (Culberson and Schaeffer, 1998) are abstraction heuristics. They are obtained by abstracting away significant portion of the problem that is small enough to be solved optimally for every state. The results are stored in the database (Edelkamp, 2001; Haslum et al., 2007).

– *Critical Paths* heuristics map the shortest-path planning problem in state space into shortest-path problem in *atom space* (a simplified problem) (Geffner and Haslum, 2000; Haslum and Geffner, 2001).

– *Landmarks* are implicit subgoals that must be achieved when achieving top level goals of the problem (Richter et al., 2008). Landmarks were first introduced by Porteous, Sebastia and Hoffmann (Porteous et al., 2001).

2.5.1 Classical Planning Heuristics

For more than a decade, various classical planning heuristic approaches have been proposed, including the additive and the max heuristics (Bonet et al., 1997; Bonet and Geffner, 1999, 2001)and landmark heuristics (Hoffmann et al., 2004; Helmert et al., 2007; Richter et al., 2008; Karpas and Domshlak, 2009; Richter and Westphal, 2010). Ivankovic et al. recently extended the additive heuristic to reason about lifted parameters (i.e. infinite domain action parameters) that appears in the classical planning problems (Ivankovic et al., 2014). In this section, we briefly describe relevant heuristic approaches for classical planning problems. These heuristics are extended to handle temporal and numeric planning problems, which will be discussed in further sections.

2.5.1.1 The Planning Graph

Blum and Furst developed a compact graph structure called *the planning graph* used in the Graphplan system (Blum and Furst, 1997). Although the planning graph is not a heuristic method, it became the core of many successful heuristics in planning, including the relaxed planning graph (Hoffmann and Nebel, 2001) and its successors (Hoffmann, 2003; Hoffmann et al., 2004; Coles et al., 2009c). A planning graph is a directed and layered graph with two types of nodes and three types of edges. The layers of the graph alternate between *proposition layers* containing proposition nodes and *action layers* containing action nodes. The first layer is a proposition layer, *fl(1)*, containing all propositions (or facts) that are true in the initial conditions. The second layer is an action layer, *al(1)*, consisting of applicable action nodes based on true facts in the first layer. The third layer is a proposition layer, *fl(2)*, consisting of possibly true facts, and so on. The graph expansion continues until it reaches a proposition layer satisfying all the goals of the problem. Then, a recursive backward search algorithm is invoked starting from this proposition layer. For each fact to be achieved, the planner selects achieving actions (of these facts) in the previous action layer one after another, which are not exclusive (discussed in the following two paragraphs) of any other action that has been already selected.

The edges in a planning graph link the proposition and action layers. For $i \geq 1$, action layer i and proposition layer i are linked with *precondition edges*. Action layer i and proposition layer $i+1$ are linked with *add-edges* denoting the add effects and *delete-edges* denoting the delete effects of the actions in action layer i. Figure 2.4 illustrates the planning graph of a simple logistics problem, where there are two locations, A and B, and a truck Tr (initially at A) the package Pk (initially at B). The goal is to have the package at location A. The action *(move Tr B A)* becomes applicable at action layer *al(2)*, as all of its conditions hold in the fact layer *fl(2)* (i.e. it requires the fact *(at Pk B)* to hold). The effects of this action delete the fact *(at Tr B)* from and adds the fact *(at Tr A)* to the fact layer *fl(3)*.

An important feature of the planning graph is noticing and propagating *mutual exclusion (mutex)* relations among nodes. Two actions, a and b, are marked as exclusive of each other if a deletes a precondition or adds an effect of b; or two mutex propositions p and q are preconditions of a and b, respectively. Bonet and Geffner describe the Graphplan system as a heuristic search planner with a heuristic function and a standard search algorithm (Bonet and Geffner, 1999). The heuristic value at a state s, $h^G(s)$, is derived from the index of the first layer in the graph that includes all the propositions in s without a mutex.

The Graphplan system stores a set of subgoals that are deemed unsolvable at time t in a hash table. The system probes the hash table to see if the new set of subgoals at time t has been previously deemed unsolvable before searching it. This technique is called *memoisation*, that speeds up the search and allows the system to provide termination guarantees.

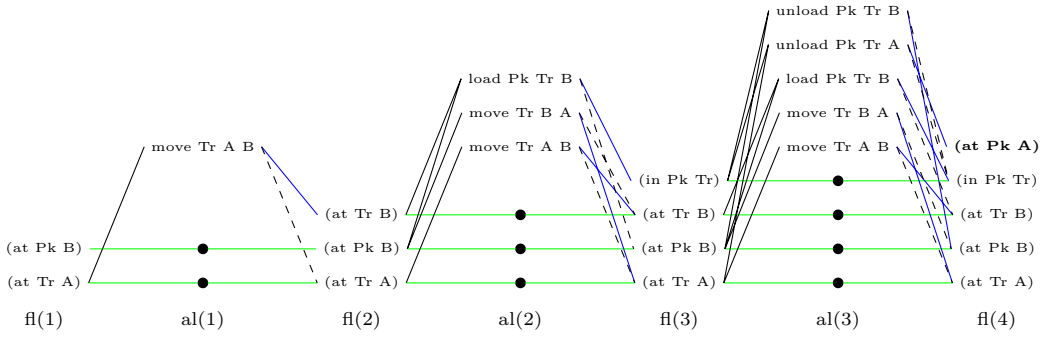


Figure 2.4: The planning graph for a simple logistics example. The black dots denote no-ops (i.e. no operation nodes), black solid lines denote conditions, blue solid lines denote the add effects and the dashed lines denote the delete effects.

2.5.1.2 The Relaxed Planning Graph Heuristic

Hoffmann and Nebel proposed using the relaxed version of the planning graph as a heuristic (Hoffmann and Nebel, 2001). The relaxation comes from ignoring delete effects of actions (the delete-relaxation). The resulting heuristic is called Relaxed Planning Graph (RPG) heuristic. The heuristic does not contain any mutex relations, and consequently, it reaches a solution without backtracking. The heuristic applies the graph expansion and solution extraction techniques as usual and returns the heuristic distance to the goal ($h^{FF}(s)$) and a list of helpful actions. The heuristic distance is computed based on the number of actions chosen during solution extraction. The helpful actions are a subset of

these chosen actions that appear at the first action layer of the relaxed planning graph (if this relaxation is applied to the example in Figure 2.4, *(move Tr A B)* action would be the only helpful action in the RPG at state s).

2.5.2 Numeric Planning Heuristics

The complexity of numeric planning is undecidable and it can have infinite state propagation. Hence, it is crucial to use a heuristic providing *basic informedness* in polynomial time (at worst). The basic informedness is an important property of a delete-relaxation, which refers to whether the heuristic is capable of generating a relaxed plan that achieves all the preconditions and goals of the original problem, or not. More formally, Hoffmann defines it in numeric planning task context as follows:

Suppose that (V, P, A, I, G) refers to the numeric planning task $\langle \mathbf{v}, F, A, I, G \rangle$.

> "**Basic Informedness:** the preconditions and goals can trivially be achieved in the original task if and only if the same holds in the relaxed task. Assuming a restricted numeric task (V, P, A, I, G), $\langle\rangle$ is a plan for (V, P, A, I, G) if and only if $\langle\rangle$ is a relaxed plan for (V, P, A, I, G), and for $a \in A$, $result(I, \langle\rangle) \models pre(a)$ if and only if $result(I, \langle\rangle) \models pre(a^+)$" (Hoffmann, 2003, pg. 295 and pg. 301).

In this section, we study an extension of the delete-relaxation heuristic of the FF to numeric planning problems. Then, we survey a hybrid heuristic approach that exploits a *mathematical solver* (alongside the RPG heuristic) to compute strong bounds of variables during constraint propagation. Finally, we talk about a landmark-based numeric planning heuristic that aims to tackle numeric subgoaling problems.

2.5.2.1 The Metric-RPG Heuristic

The Metric Relaxed Planning Graph (The Metric-RPG) heuristic has been the most popular numeric planning heuristic (based on the research output in numeric planning) for more than a decade (Hoffmann, 2003). It extends the propositional delete-relaxation of the RPG heuristic used in the FF planner to numeric variables. The numeric expressions are transformed into a restricted task, called *the linear numeric task*, in which the arithmetic operations are restricted to $\{\geq, >\}$ in conditions and $\{+=\}$ in effects. The resulting form, called *Linear Normal Form (LNF)*, ensures that the numeric variables grow in only one direction.

In the case of propositions, the heuristic ignores propositional delete effects (same as in the RPG heuristic). In the case of numeric variables, applying relaxed actions diverges the bounds by decreasing the lower limits with decrease effects and increasing the upper limits with increase effects. The bounds of the numeric variables at a layer are computed as follows. Suppose that $v \in \mathbf{v}$ denotes the numeric variable v in a vector of numeric variables $\mathbf{v}$, i denotes the index of the current layer, $v_{max}(i)$ and $v_{min}(i)$ denotes the upper and lower bounds of v, $a_{max} \uparrow (i, v)$ and $a_{min} \downarrow (i, v)$ denotes the maximum and minimum values of all possible assignment effects on v, $increase(i, v)$ and $decrease(i, v)$ denotes the increase and decrease effects on v at layer i, respectively. Then the bounds of v at the next layer $i + 1$ become:

$$v_{max}(i+1) = max\{a_{max} \uparrow (i,v), v_{max}(i) + \sum increase(i,v)\}$$
$$v_{min}(i+1) = min\{a_{min} \downarrow (i,v), v_{min}(i) + \sum decrease(i,v)\}$$

The numeric precondition satisfaction is simply done by checking each precondition whether it is satisfied by any value within the diverging bounds. The heuristic applies the graph expansion to check whether all numeric and propositional goals become reachable and the solution extraction to compute the heuristic distance to the goal, $h(s)$, and a list of helpful actions in order to guide the search. The graph expansion phase is also called *the reachability analysis*.

2.5.2.2 The LP-RPG Heuristic

Although the delete-relaxation based heuristics are extremely powerful on guiding the search in numeric planning problems, they become less informative when predicting the true bounds (i.e. numerically feasible bounds) of variables. As the graph expands in the Metric-RPG heuristic, the relaxed bounds of variables are computed: decrease effects do not update the upper relaxed bound and increase effect do not update the lower relaxed bound. As a result, the relaxed bounds of a variable get diverged from each other (covering infeasible values as well as the feasible ones) as the graph expands.

There are situations in planning problems in which the divergence of bounds in the Metric-RPG heuristic can provide wrong guidance. These problems include *resource persistance* and *helpful action distortion* (Coles et al., 2013). Resource persistence is a consequence of using the delete-relaxation. When a numeric resource is consumed it does not disappear in the relaxation of negative effects, so the resource persists. The opportunity of reusing the resource can suggest that there is a significantly shorter plan available than is the case in reality. The phenomenon of helpful action distortion emerges as a consequence of misleadingly shorter plans generated due to resource persistence. This phenomenon, the helpful action distortion, refers to the failure of capturing actions in the relaxed plan that are essential towards meeting the goals of the planning problem. This problem usually occurs in case the planning problem requires repetitive numeric transfer (a numeric resource is repetitively decreased (or consumed) and increased (or produced)), such as in the Settlers domain (Long and Fox, 2003). The relaxed plans typically contain too few actions that produce the resource (i.e. production actions). In this situation, production actions often do not appear in the helpful action set, and are not included in local search (e.g. EHC). As a consequence, the heuristic can promote false activities as helpful actions, which can lead the search to a dead-end.

The most recent work focusing on tackling the weaknesses of the delete-relaxation (e.g. helpful action distortion and resource persistence) using a mathematical solver is *the LP-RPG heuristic* (Coles et al., 2013, 2008b). The heuristic uses linear programming (LP) solver alongside the Metric-RPG heuristic. Although the heuristic generates an informative relaxed plan, it suffers from intensive use of LP-solving in order to obtain strong variable bounds. The number of LP calls per RPG graph is equal to the multiplication of the number of state variables, actions and layers: the heuristic optimises the upper and lower bounds of each numeric variable of each action at every action layer.

2.5.2.3 Numeric Landmarks Heuristic

The most recent work on landmarks (and in fact, a parallel work to the thesis) is the work of (Scala et al., 2017) extending the concept of propositional landmarks to numeric landmarks, which are numeric conditions that must be achieved in an attempt to satisfy top-level numeric goal conditions. The work is based on subgoaling relaxation (Scala et al., 2016b). It exploits AND/OR graphs (Keyder et al., 2010) to capture numeric and propositional subgoal dependencies and relies on solving LPs. The work results in an admissible heuristic for cost-optimal numeric planning problems. This work is based on subgoaling relaxation and focusses on solving generalised numeric planning problems. Other researchers have also surveyed solving generalised numeric planning problems using interval-based relaxation approaches (Aldinger et al., 2015; Scala et al., 2016a). These approaches propose solutions to generalised numeric tasks (supporting all arithmetic operations). To overcome the complexity of the generalised tasks, the Metric-RPG heuristic transforms numeric tasks into the LNF tasks. Since then, it has been the underlying principle of numerous numeric planning systems for expressive PDDL languages, such as POPF (Coles et al., 2010) and Scotty (Fernández-González et al., 2015a) planner families.

In Chapter 4, we investigate numeric producer-consumer relationship between actions (i.e. numeric resources are exchanged for one another), showing that this relationship generates numeric subgoals, which are a sort of numeric landmark when achieving top-level goals. The heuristic approach we propose in this chapter relies on the LNF transformation.

2.5.2.4 The Interval-Based Relaxation

Reasoning with the numeric intervals have been thoroughly studied in the fields of applied mathematics and interval arithmetic for decades (Young, 1931; Moore et al., 2009). In the interval-based relaxation (IBR), each numeric variable at a state is mapped into an interval of real values denoting the set of values the variable can take. The Metric-RPG heuristic fits to the class of IBR based on this definition. However, the transformation to the LNF restricts reasoning with the complete scope of the numeric planning problem. This restriction eventually yields a subset of the general numeric problem.

The first instance of using interval-based methods in general (or unrestricted) numeric planning is relatively new. Aldinger et al. recently studied using this method in relaxation theoretically (Aldinger et al., 2015). The authors propose that solving numeric planning tasks reasoning with all arithmetic base operations $(\times, \div, +, -)$ is decidable. However, their solution fails solving most numeric problems with cyclic dependency between variables. Scala et al. recently surveyed the generalisation of the IBR to handle non-linear constraints and cyclic behaviour between variables (Scala et al., 2016a).

2.5.3 The Temporal Relaxed Planning Graph (TRPG) Heuristic

The Temporal Relaxed Planning Graph (TRPG) (Coles et al., 2009c, 2008a) heuristic extends the RPG heuristic to include time. It considers action durations and manages the start-end semantics of PDDL2.1 (Fox and Long, 2003). The graph expansion and the solution extraction of the TRPG is similar to the RPG, except that each fact layer is assigned with the earliest timestamp that can be reached (instead of a step index). New snap-actions

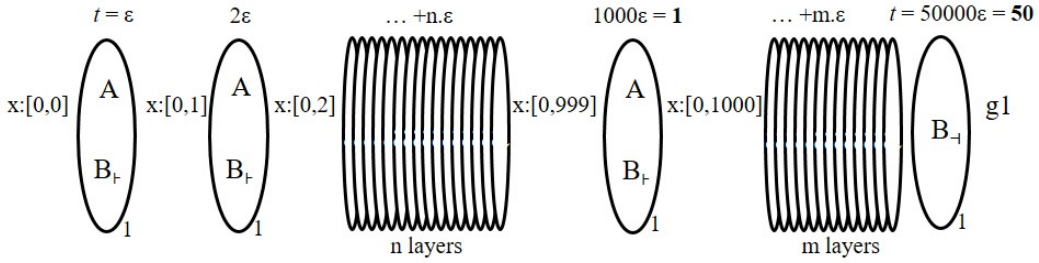


Figure 2.5: Schematic representation of the discretised graph expansion.

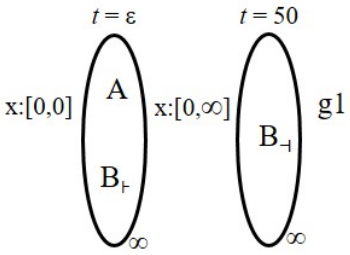


Figure 2.6: The illustration of the infinity analysis.

(each durative action is split into two snap-actions denoting the start and the end of action to reason with the start-end semantics of PDDL2.1) and facts are delayed until the earliest times at which they can appear.

2.5.3.1 The Infinity Analysis

There is an additional important feature of the TRPG that differs from the RPG during graph expansion: when a snap-action is applicable at an action layer (where all numeric and propositional conditions of this action are satisfied), it is relaxed to have arbitrarily many concurrent copies at the same layer. As a result, the effects of this action are applied arbitrarily many times in parallel (at the same layer), and this makes the reachable range of the affected numeric variables become unbounded. Hoffmann (Hoffmann, 2003) observed this possibility and referred to it as ∞ *handling technique* and ∞ *trick*, but in the Metric-RPG, Hoffmann used an incremental extension of the bounds: allowing actions to be applied only once per action layer, which carries a risk of building exponential number of relaxed planning graph layers for a search state. We refer to this concept as the *infinity analysis* in this thesis.

In the TRPG the incremental approach cannot be used practically, because the ability to apply an action repeatedly in tiny time increments blows up the reachability analysis. Therefore, the TRPG must use infinity analysis to be practical.

To illustrate why using the incremental approach (as in the Metric-RPG) is impractical in the TRPG, consider the following temporal-numeric problem. Suppose that x is a numeric variable and $g1$ is a propositional fact. B is a durative action (its duration is 50). The end snap-action of B has a numeric condition of $x \geq 1000$ and adds $g1$. A is a non-durative (instantaneous) action that increases x by 1. The fact $g1$ initially does not hold and the goal

is to achieve it. Suppose the layers of the reachability graph is incremented by $\epsilon = 0.001$ time units. Then, the approach repeatedly applies action A and extends the graph 1000 layers (that is equivalent to 1 time unit) to satisfy $x \geq 1000$ condition. The approach expands the graph for another 49000 layers as the end of B appears at $t = 50$. Figure 2.5 illustrates this situation. It is clear that generating countless layers can consume significant portion of the memory and it can be extremely harmful to the efficiency of the heuristic, whereas it can be handled with a few layers. Figure 2.6 shows the graph expansion using the infinity analysis. The instantaneous action A and the start of action B appear at the earliest layer, $t = \epsilon$, as they do not have any condition that must hold. The numeric effects of these actions are considered to be applied arbitrarily many times at this layer. As a result, the variable x becomes immediately unbounded (i.e. $x : [0, \infty]$), which satisfies the numeric condition at only one layer. Then, the next action layer becomes $t = 50$, as it is the earliest time the end of action B can appear.

2.6 Brief Descriptions of the Planning Systems

In this section, we describe a wide range of planning systems that are closely connected to the thesis. We thoroughly describe the temporal-numeric planners that our system POPCORN is built on (e.g. the POPF planner), and other state-of-the-art planners that can reason with control parameters (e.g. the Scotty planner). We briefly mention other popular planning systems by generalising based on their planning representations as well as the methods they use.

2.6.1 The POPF Planner and Its Predecessors

The POPF planner (Coles et al., 2010) is an informed forward search temporal planner that avoids early commitment to the action orderings and it encodes plans as partially specified plans. The rest of the features of the planner (e.g. temporal-numeric reasoning and handling continuous numeric effects, and so on) are inherited from its predecessors FF (Hoffmann and Nebel, 2001), CRIKEY3 (Coles et al., 2009c) and COLIN (Coles et al., 2009b, 2012). In this section, we briefly describe these systems in the chronological order they are released. It is worth mentioning that POPF preserves all of the reasoning capabilities of its predecessors.

– **Fast Forward (FF) and Metric-FF** The FF planner is developed by Hoffmann and Nebel that relies on informed forward state space search using the RPG heuristic (Hoffmann and Nebel, 2001). It exploits EHC as its main search algorithm. If the EHC fails, it resorts to best-first (weighted A^*) search. The planner makes use of helpful actions to prune less promising states in the EHC. Figure 2.7 shows the relationship between the search and the heuristic in the FF system. The same authors extended the FF planner to reason with linear conditions and discrete numeric change, resulting in a planner called Metric-FF (Hoffmann, 2003). The Metric-FF planner performs the same heuristic search strategy as in the FF. It extends the RPG heuristic to support the computation of a relaxed plan for problems involving discrete numeric change.

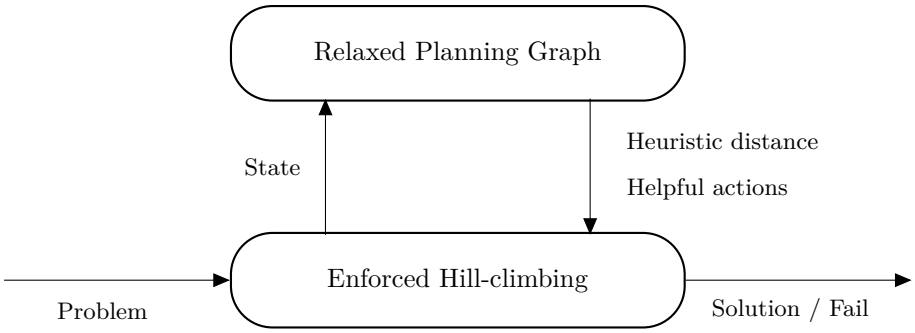


Figure 2.7: The illustration of the FF planner framework. Reproduced from (Hoffmann and Nebel, 2001).

– **CRIKEY3** (Coles et al., 2009c) is a forward state space search temporal planner that extends the Metric-FF planner to reason with time. It also extends the Metric-RPG heuristic (the heuristic of the Metric-FF) to include temporal structure of the plan (resulting in the TRPG heuristic presented in Section 2.5.3). The planner splits durative actions into start and end *snap-actions* in order to respect the semantics of PDDL2.1 language. Each state s in the search space consists of the valuations of the sets of the propositional facts and the numeric variables, the ordered list of actions that are already started but not yet finished (i.e. only the start snap-action is applied) and a collection of temporal constraints over the actions that appears in the plan to reach s. The planner exploits Simple Temporal Network (STN) (Dechter et al., 1991) to evaluate these temporal constraints accumulated from successive action choices at a state.

– **COLIN** (Coles et al., 2009b, 2012) is an informed forward state space temporal-numeric planner that can reason with linear continuous effects exploiting full semantics of PDDL2.1. It uses total ordering as its branching scheme. It is built on CRIKEY3 planner (Coles et al., 2008a) and uses linear programming to evaluate the temporal and numeric consistency of the state reached. It extends the heuristic used in CRIKEY, the temporal RPG, to provide guidance in problems with continuous effects. The relationship between discrete and continuous changes are modelled similar to the hybrid automata theory. Figure 2.8 illustrates the relationship from the aspects of state transition and time dimensions of applying an abstract action A. Any discrete effect on variable v at $step_i$ updates its value from v_i to v_i'. Any continuous effect on v between $step_i$ and $step_{i+1}$ continuously updates its value from v_i' to v_{i+1}. Suppose that Δv_{now} denotes the accumulated rate of continuous change on v between time $step_i$ and $step_{now}$. The variable v_{now} gives the value of $\mathbf{v}$ between the steps and it is computed as follows:

$$v_{now} = \Delta v_{now} \times (step_{now} - step_i)$$

It is worth noting that based on the semantics of continuous numeric change, the duration of a durative action is a free variable while the rate-of-change remains

constant. This strategy is opposite to the approach of Kongming planner (Li and Williams, 2008), which is introduced further in this section.

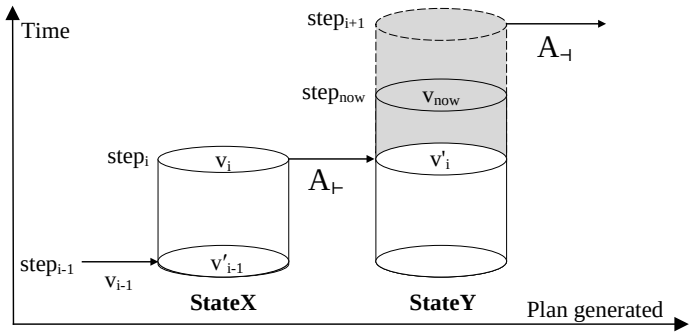


Figure 2.8: Schematic representation of discrete and continuous effects on variable v.

– **POPF** (Coles et al., 2010) planner extends COLIN to support partial order planning. It delays the commitment to action ordering decisions and the timestamps and the valuation of variables. This concept increases the flexibility of the planner in execution. In case the problem has discrete or continuous time-dependent change, the planner uses linear programming to check numeric and temporal consistency; otherwise it uses STN for the same purpose. The planner exploits a slightly modified version of the TRPG used in COLIN, where each fact can only appear after its achiever (or re-achiever) action appears in the graph. The POPF planner has another feature called the *anytime mode*. When it is enabled, the planner looks for a state that yields the best possible solution (with respect to the metric objective function, or the makespan by default) until all states in the search space are being explored. As a numeric planning problem can yield to infinitely many states, the planner does not terminate until it reaches the memory or time limits. As we also reason about planning problems with complex branching factors in this thesis, the termination criterion of our system, POPCORN, is identical to that of POPF.

2.6.2 Planners Reasoning with Control Parameters

Work exploring the use of control parameters in domain-independent planning is limited. The duration parameter was introduced in PDDL2.1 (Fox and Long, 2003), but very few planners can plan with actions that have both flexible durations and duration-dependent effects. Examples that can include POPF (Coles et al., 2010) and its close cousins, COLIN (Coles et al., 2009b) and OPTIC (Benton et al., 2012). In addition, some PDDL+ planners (that are not forward heuristic search planners) can also reason with flexible durations, which are UPMurphi (Penna et al., 2009) (and its relative DiNo (Piotrowski et al., 2016)) and SMTPlan+ (Cashmore et al., 2016). LPSAT (Wolfman and Weld, 2001) and TM-LPSAT (Shin and Davis, 2005) use a hybrid of LP and a combinatorial SAT-solver to plan but these approaches greatly suffer from the absence of heuristic guidance.

The authors of the Kongming (Li and Williams, 2008) and the Scotty planners (Fernández-González et al., 2015a,b) investigate planning with control parameters in a

domain-independent way. Pantke et al. investigate this concept in the production control application scenario. We describe these systems in detail in this section.

2.6.2.1 Kongming

Kongming is a Graphplan-based hybrid planner, which can reason with numeric parameters that takes their values from infinite domains, and they call them *control variables*. The planner uses an extended PDDL version that allows encoding such parameters in the domain model. The language used by Kongming will be presented in Chapter 3. The planner expands the planning graph (each layer in the graph is called a *happening*) to handle the discrete causal relationship between actions and captures the interaction of dynamic continuous action variables with the *flow tube* representation. The resulting planning graph is called *Hybrid Flow Graph*. The flow tubes contain the trajectories of the variables as the graph expands over time through the planning graph. The graph is then translated into a mixed integer linear programming (MILP), and solved using an optimisation tool (CPLEX). The use of Graphplan restricts the representation of time, so it is discretised, while the rates of change of process effects are taken as control variables. This approach contrasts with COLIN, POPF and TM-LPSAT where the duration is taken as a variable and rates of change remain constant (over intervals). It is also worth noting that Kongming does not allow defining flexible duration bounds for durative actions (all durative actions must be defined to have fixed durations). Kongming suffers from excessive number of happenings generated (due to deepening search) that is interleaved with an optimisation tool.

The trajectory of a variable (R_g) at a succeeding layer (or a happening) are computed using the feasible region of the variable (R_i), the duration of the action d and approximated dynamics of the variable (the gradient change) dB. Figure 2.9 shows how two flow tubes are connected when expanding the planning graph. The circular area in green shows the intersection of the ends of the flow tubes. In order to add a new action (e.g. $a2$) to the plan the intersection between the end and start of two flow tubes must correspond to a non-empty cross section; otherwise the continuous flow will not occur. Since this approach is unsound, the planner uses MILP solver to confirm the validity of the plan.

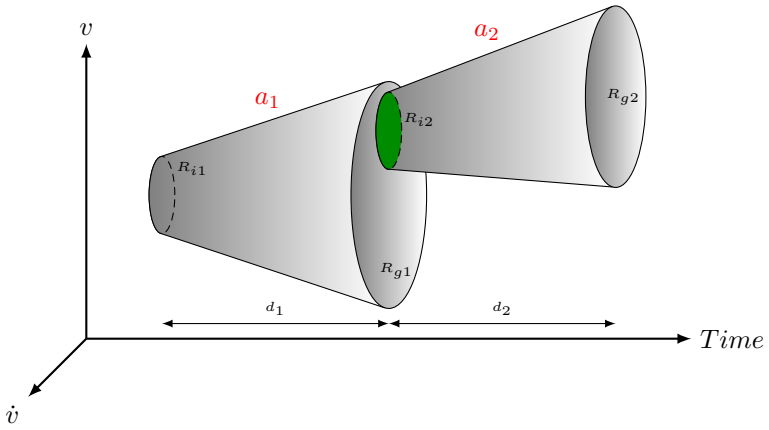


Figure 2.9: Schematic representation of connecting the flow tubes of actions a_1 and a_2.

2.6.2.2 Scotty and cqScotty

Fernández-González, Karpas and Williams have recently studied planning with infinite domain action parameters (Fernández-González et al., 2015a). They refer to these parameters as *control variables*. These parameters are formulated as flexible rates of change in continuous effects. Their work outlines the development of the Scotty planner. It combines the flow tube representation of Kongming with the forward-chaining search and the linear programming used in the COLIN planner. The flow tubes are used to capture continuous effects with control variables. It uses forward state space search to overcome the limitation on numbers of happenings in Kongming.

Note that the Scotty planner does not support actions performing discrete numeric change (Fernández-González et al., 2015b), so that all numeric changes captured in an action is *duration-dependent* (including the numeric change with a control variable, where it is a flexible rate of change over the duration of the action). In this thesis, we investigate planning with such parameters in a different way. In our approach, the domains of these parameters are not necessarily dependent on each other and they are independent from the duration.

The same authors recently extended Scotty to transform non-convex constraints to a convex linear over approximations to handle non-linear quadratic effects, and uses a Second Order Cone Programming tool to check consistency (Fernández-González et al., 2017). In this work, the planner suffers from the fact that the heuristic can promote false activities in certain conditions, such as resource transfer with control variables. In Chapter 4, we introduce a generalised approach to tackle this issue for forward heuristic search planners.

The purpose of control parameters in the Kongming and the Scotty planners (and earlier work using flow tubes (Hofmann and Williams, 2006)) is to convey flexibility of choice to the *executive* (Effinger et al., 2009). Therefore, Scotty finds *a flexible plan*, in which it leaves the decision of the values of control parameters and the execution times of actions to an executive during plan execution (even though it can make choices that invalidate the plan). We describe how the planner obtains a flexible plan below.

In Scotty, all control parameters represent flexible rates of change in continuous effects, which are restricted to be *linear* and only include a single control parameter. Each continuous effect with a control parameter is formalised as $\langle x, d, k \rangle$ that is active between the start and the end times of the continuous effect (t_{start} and t_{end}, respectively), where x is the state variable, d is the control parameter and k is a constant. The value of the state variable x at time t, is computed by $x(t) = x(t_{start}) + k \cdot d \cdot (t - t_{start})$, where $t_{start} \leq t \leq t_{end}$. It is clear that this formulation yields a non-linear interaction between the flexible rate of change (i.e. the control parameter d) and the continuously evolving time variable (i.e. $(t - t_{start})$, which is denoted by `#t` in the PDDL2.1). In order to avoid this non-linearity, the planner assigns a fixed value, $d_{var}(t)$ at time t, to the control parameter d during the execution of the continuous effect, where $d_{var}(t)$ is in between the upper and lower bounds of d. This process results in generating *a fixed plan*, which consists of a sequence of timestamped actions with fixed control parameter values. Since a fixed plan is not robust in execution, the planner extracts a flexible plan from the fixed plan using Qualitative State Plans (QSPs) (Léauté and Williams, 2005) network and reports it to the executive. When traversing from a fixed

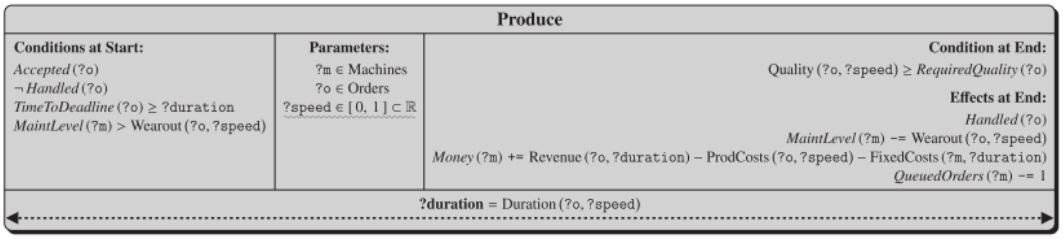


Figure 2.10: An example of continuous numeric action parameter encoding in proposed PDDL language given in Mechatronics Journal (Pantke et al., 2016).

plan to a flexible one, the timestamps of actions and the values of control parameters are replaced with flexible bounds so that the executive have freedom to choose the execution times of actions and the control parameter values. It is also worth noting that QSP networks exploit CPLEX as the underlying solver to find these flexible bounds.

In common with most planning systems that exploit delete-relaxation based heuristics, the Scotty and the cqScotty planners also suffer from the helpful action distortion issue, especially in producer-consumer problems where numeric transfer is achieved by control parameters. These planners fail to recognise actions replenishing consumed resources as helpful and rely on intensive backtracking to find a solution or run into a dead-end. In Chapter 4, we introduce an extension to the TRPG heuristic that resolves the issue in these problems.

2.6.2.3 Two-stage Production Planning and Control

Pantke, Edelkamp and Herzog have recently presented a PDDL-based multi-agent non-linear planning approach to reason with infinite domain action parameters in production planning and control (Pantke et al., 2014, 2016), and they call them *continuous numeric action parameters*. The authors propose an extension to PDDL language that allows encoding these parameters in the existing `:parameters` field in PDDL action. Figure 2.10 illustrates the example encoding of their proposed PDDL extension. In this example, the `?speed` is an infinite domain action parameter where it is allowed to take any real-valued number between [0,1]. Observe that the parameter is typed with $\mathbb{R}$ symbol in this encoding, yet it remains unclear how this *reserved* type description can be parsed or encoded in the PDDL domain model.

The authors exploit two-stage planning strategy in their work. First, the planner finds partially grounded total-order plans with *lifted* infinite domain action parameters. Then, the planner finds the fully grounded plans by determining the bounds of the variables of the first n partially grounded plans (using a mathematical optimisation tool) based on their potential quality. The optimisation tool is not used to check the consistency of a state. Instead, they use the *interval arithmetic* relaxation, which gives the possible values (not the optimal values) at a state. Additionally, the planning horizon is fixed and they do not use any relaxation heuristic during search.

Although the proposed PDDL extension can be used by fully domain-independent planning systems, the proposed methodology is not analysed in a domain-independent way. Instead the plan enumeration (and generation) has been adapted to the problem setting,

which is indirectly enforced by an industrial application. We also believe that this work can suffer from poor scalability in a more complex planning scenario due to lack of heuristic in search. In this thesis, we seek fast and scalable domain-independent solutions for this planning field, which can be easily exploited by any other forward heuristic search planner.

Our motivation in introducing control parameters is slightly different to that of Williams et al. and Fernández-González et al.: we are interested in the planner and the executive choosing values to construct a fixed plan *before the execution*. We want to reach a correct plan and we want a plan that is efficient. Leaving constraint resolution to plan-execution risks that the constraints cannot be resolved at that time, so our approach proves the existence of the plan by finding an explicit resolution of the constraints at planning time. Our control parameters do not necessarily represent rates of change (for example, the cash withdrawal), and effects of actions can be independent of duration, or not. POPCORN can capture rates of change as long as duration is fixed. The work of Pantke et al. (Pantke et al., 2014, 2016) focusses only on a specific production control domain application, whereas we consider the domain-independent planning applications in forwards search framework using relaxation-based heuristics.

2.6.3 Other Planners

Several planners exploit Graphplan, including TPSYS (Garrido et al., 2001, 2002) that can solve required concurrency problems, LPGP (Long and Fox, 2003) that uses linear program to aid choosing actions in the graph, and the work of (Huang et al., 2009) that translates the graph to a SAT formula. On the other hand, various planners use forward state space search to reach a goal, including HSP (Bonet and Geffner, 2001) and SAPA (Do and Kambhampati, 2003) planners, VHPOP (Younes and Simmons, 2003) that is a partial order planner, FD (Helmert, 2006) planner that governs multi-valued representation (i.e. *SAS+* formulation) of the classical planning task, TFD planner (Eyerich et al., 2012) that extends FD to reason with durative actions. A few hybrid planners governs the *discretise and validate* approach and exploits *planning as model-checking* paradigm (Cimatti et al., 1997; Bogomolov et al., 2014), including UPMurphi (Penna et al., 2009) and DiNo (Piotrowski et al., 2016) that extends UPMurphi planner to get heuristic guidance.

2.7 Summary of Temporal-Numeric Planners

We described various planning systems in this chapter. Here we summarise the capabilities of the most relevant approaches in Table 2.1. The important feature is handling *infinite domain action parameters*, which is recently achieved by a few planners. Observe that none of the domain-independent systems can handle duration-independent parameters. We introduce our approach that fills this gap while preserving the highest domain-independent planning compatibility (i.e POPF system) and at the same time reasoning with a greater PDDL expressivity in this thesis.

Table 2.1: Comparison of the capabilities of the most recent temporal-numeric planners.

							Infinite Domain Action Parameters	
Planners	Domain Indep.	Temporal	Numeric	Duration Inequal.	Required Concur.	Partial ordering	Duration Indep.	Duration Dep.
LP-RPG	✓		✓		✓			
TPSYS	✓	✓	✓		✓			
LPGP	✓	✓	✓					
SAPA	✓	✓	✓		✓			
CRIKEY3	✓	✓	✓		✓			
TFD	✓	✓	✓	✓	✓			
COLIN	✓	✓	✓	✓	✓			
POPF	✓	✓	✓	✓	✓	✓		
TM-LPSAT	✓	✓	✓	✓				
UPMurphi	✓	✓	✓	✓	✓			
DiNo	✓	✓	✓	✓	✓			
Kongming	✓	✓	✓	✓				✓
(cq)Scotty	✓	✓	✓	✓				✓
Pantke et al.		✓	✓	✓			✓	✓

2.8 Integrated Task and Motion Planning (TAMP)

Over the last decade, a significant progress in ensuring the geometric feasibility of symbolic plans in highly confined workspaces has been made by various researchers (Srivastava et al., 2014; Plaku and Le, 2016). Other pioneering approaches can be summarised as follows.

aSyMov (Cambon et al., 2009) is a highly recognised TAMP planner that integrates the Metric-FF task planner with a motion planner. It extends the Metric-FF to check whether the symbolic action to be applied results in a geometrically feasible state (based on the feedback from motion planner). Although the Metric-FF is a domain-independent planner, extending it to check the state geometric reachability makes the approach domain-specific. A parallel work to this approach is recently studied by Garrett, Lozano-Perez and Kaebeling (Garrett et al., 2015, 2018), and their system is called *FFRob*. In their work, they identify that aSyMov suffers from the lack of heuristic guidance, thus they propose an extension to the RPG heuristic in the FF planner that takes geometric information into account.

Erdogan and Stilman (Erdogan and Stilman, 2013; Erdogan, 2016) encode geometrical constraints in action descriptions (in PDDL-like representation) and propagate the constraints while finding a symbolic plan for a structural design problem. Although the approach reasons with action level constraints, the task planner is not domain-independent.

Erdem et al. (Erdem et al., 2011), (similar to the approach of Dornhege et al. (Dornhege et al., 2009)), use a fully symbolic task planner. In their approach, the task and motion planner guides each other in case any of these planners deem unsolvability. For instance, in case the task planner cannot find a symbolic plan, the motion planner updates the problem file for re-planning. If the motion planner fails finding a trajectory, the task planner suggests a new symbolic route to the motion planner.

In our work, we achieve symbolic plans that are aware of the geometrical constraints encoded in *a domain-independent PDDL environment*, so that any domain-independent task planner that can reason with the full semantics of the PDDL2.1 language and our proposed language extension (presented in Chapter 3) can be used. Unlike aSymov and FFRob, the

geometric constraints, which model the locations as continuous regions, are directly encoded in the action schemas.

Chapter 3

Planning Using Actions with Control Parameters

3.1 Introduction

PDDL is a powerful modelling language, but one limitation of the models it supports is that action parameters are only allowed to be drawn from finite domains. These domains are enumerated in the problem descriptions and are typically relatively small (a few tens of objects would already be unusual). A consequence of this constraint is that action schemas can be grounded, by substitution of parameters with values in all possible ways, to yield a finite set of actions for each problem instance. There is one parameter that is an exception to this: the duration of a durative action may be left flexible, so that the planner can choose its value (possibly subject to constraints). In fact, this is a far-reaching choice, since in combination with duration-dependent effects, it can make it possible to model domains in which there is an infinite branching choice of actions applicable in a state. Although the addition of numeric state variables to the classical propositional language allows construction of an infinite state space (as discussed in Section 2.2.2), without the flexible duration parameter, the state space is always *locally finite*, which is to say that only finitely many states are reachable from a given initial state, using plans bounded by a given finite length. The introduction of the flexible duration parameter changes this, so that infinitely many states may be reachable by plans even bounded to length one. This observation makes clear that the introduction of a flexible duration parameter fundamentally changes the possible structure of state spaces. Figure 3.1 illustrates this in a small example where the duration parameter is fixed or flexible in the forwards search framework. Each circular node represents a grounded state and the dashed elliptical node represents a partially-grounded state that can consist of infinitely many grounded states.

In this chapter, we consider a generalisation of the duration parameter, allowing actions to take multiple parameters from infinite domains. We call these parameters *control parameters*, since they often represent physical control parameters that a planner might select in order to make an action to have a desired effect (or intensity of effect). Examples include volume settings, velocities, power use, volumes or sizes. As a concrete example,

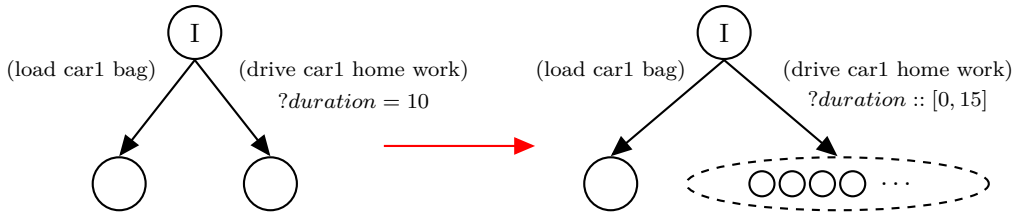


Figure 3.1: The illustration of the effects of having flexible duration parameter in state space.

illustrating that duration is not always an appropriate surrogate for control parameters, consider the action of withdrawing cash from an ATM (or cashpoint). Execution of this action involves choosing the value of the withdrawal. Although there is probably some small element of the duration of a withdrawal that is affected by the amount withdrawn, the transaction is, for all practical purposes, best modelled as a constant duration action that has an effect on cash in pocket determined by the selected value of the control parameter: the size of the withdrawal. Current standard PDDL dialects require that either the action be modelled artificially, using duration to substitute for the control, or else that the choice of withdrawal amounts be restricted to some fixed and finite menu of choices.

A slightly more complex example, generalising an example drawn from related work on Kongming (Li and Williams, 2008), is a navigation action for an autonomous underwater vehicle (AUV), equipped with independent motors to drive horizontal movement (the x-direction) and vertical movement (the y-direction). In choosing a single leg of movement, the action requires selection of an x-velocity, a y-velocity and a duration. It is clearly not possible in PDDL to represent choices in this 3-dimensional choice space using a duration alone.

In this chapter, we explore the cashpoint example in more detail to illustrate the problems introduced by the use of control parameters. We then propose a way in which planning can be performed in domains containing actions with these parameters, implemented in our planner, POPCORN (Partial-Order Planning with Continuous Real Numerics), an extension of the POPF planner (Coles et al., 2010), extending and expanding the mechanisms in that planner to support control parameters beyond duration. In this extension, we retain the ability in POPF to use temporal actions, including with flexible durations and duration-dependent effects. We demonstrate that our approach scales to tackle interesting problems, considering several domains with various characteristics. Our implementation works with linear constraints, relying on the LP support in similar style to POPF, but we observe that our strategy can be generalised to different constraints, indicating how this could be achieved.

This chapter is an extended and revised version of a conference paper (Savaş et al., 2016) published by the authors at the 22nd European Conference on Artificial Intelligence (ECAI 2016). It provides more detail about the problem definition, the LP encoding and the search strategy.

The chapter is structured as follows. We survey the related work in the next section. Section 3.2 gives a more detailed description of the cashpoint motivating example. In Section 3.4 we provide technical background on the state representation in POPF, and

in Section 3.5 we describe how we extend it to implement our approach for planning with control parameters in POPCORN. In Section 3.6 we show how POPCORN scales to solve interesting problems. Section 3.7 concludes the chapter.

3.2 A Motivating Example

We now develop the simple cashpoint example to illustrate some aspects of the use of control parameters. Suppose that we are planning to go to a pub. Initially, we are at home and have only £2 in our pocket and no snacks. We aim to be at the pub with £20 in our pocket and a collection of snacks. The plan for this problem is quite simple: we must go to the cashpoint and collect cash, go to the shop to buy snacks and go to the pub. However, we only know how much cash to withdraw once the entire plan is constructed, so that we can determine how much we intend to spend. The action to withdraw cash should allow us (as planner) to decide how much to withdraw once the rest of the plan is in place.

We propose to extend PDDL2.1 (Fox and Long, 2003) so that actions may include an additional field: control parameters. These parameters are typed. We currently assume that they are numeric parameters, although our syntax supports other types. The description of the cash withdrawal action is shown in Figure 3.2. We list all control parameters, except `?duration`, in a new field, `:control`. Our language has the expressive power to state the numeric types of the control parameters. A control parameter currently can be defined as an `integer` or a `number`. In the example, the `?cash` control parameter is defined as a number in the `WithdrawCash` action. Each instantiation of the action has its own instance of this parameter and each can be given a value independently of the others. The parameter is tied into the state through action preconditions: in this case, the parameter is restricted to lie between a minimum withdrawal and the limit of the machine. It also appears in the effects, increasing what is in the pocket and decreasing what is left in the machine.

An example problem for this domain is shown in Figure 3.3. In this example there are two ATMs, each with a limited balance, and one shop at which we can buy snacks. In addition, Figure 3.3 shows the (optional) metric objective of the problem that can play an important role in the valuation of the control parameter `?cash`. The domain also includes a move action to allow movement between locations. Intuitively, to optimise the plan metric, we should withdraw sufficient cash to buy snacks and to have £20 at the pub. We would not want to withdraw more or less cash than required when at the cashpoint.

This problem could be modelled without control parameters, by discretising the amount that can be withdrawn into a finite set of possible values (say £5, £10, £20, £50). If we consider a forward searching planner, relying on a standard relaxation-based heuristic, the planner will choose a withdrawal action to meet the demands of the largest goal (the £20 goal), but the relaxation will ignore the effect of the future snacks purchase. This means, the planner will not see that it actually needs to withdraw £25, and the final plan might involve returning to the cashpoint to make a second withdrawal (the precise behaviour will depend on the search strategy, but this is what happens in POPF, for example). In the discretised model, the optimal plan (according to the metric) is to withdraw £25 using two withdrawals before moving away, while the *shortest* plan withdraws £50. When using the control parameter model, the planner can decide to withdraw exactly £23 in a single action,

```
(:durative-action WithdrawCash
 :parameters (?p - person ?a - location ?m - machine)
 :control (?cash - number)
 :duration (= ?duration 2)
 :condition (and (at start (at ?p ?a))
  (over all (at ?p ?a)) (at start (located ?m ?a))
  (at start (>= ?cash 5))
  (at start (<= ?cash (balance ?m)))
  (at start (canWithdraw ?p ?m)))
 :effect (and
  (at start (decrease (balance ?m) ?cash))
  (at end (increase (inpocket ?p) ?cash))))

(:durative-action BuySnacks
 :parameters (?p - person ?a - location)
 :duration (= ?duration 1)
 :condition (and  (at start (at ?p ?a))
  (over all (at ?p ?a)) (at start (snacksAt ?a))
  (at start (>= (inPocket ?p) 5)))
 :effect (and (at end (decrease (inPocket ?p) 5))
  (at end (gotSnacks ?p))))
```

Figure 3.2: Main actions of the cashpoint domain.

```
(:init (at Joe home) (snacksAt Store)
 (= (inPocket Joe) 2)
 (canWithdraw Joe ATM1) (canWithdraw Joe ATM2)
 (located ATM1 Bank) (located ATM2 Bank)
 (= (balance ATM1) 50) (= (balance ATM2) 100))
(:goal (and (>= (inPocket Joe) 20)
 (gotSnacks Joe) (at Joe Pub)))
(:metric minimize (inPocket Joe))
```

Figure 3.3: The initial state of the cashpoint problem.

yielding both the shortest and optimal plan.

It is worth emphasising that the POPF planner has no way of handling a non-discretised model of the cash amounts, except via the use of the duration parameter. Using the duration parameter to model the cash parameter would be problematic as the duration of the action would rely on the withdrawal amount, which would be unrealistic. For example, if we model the `WithdrawCash` action to increase the `(inpocket ?p)` variable via `?duration` (say, `(inpocket ?p) += ?duration`), we could achieve a plan to withdraw exactly £23 in a single action, but it yields a significantly longer plan as the duration of the action would have to be at least 23 time units. A more realistic option of using `?duration` would be to increase the variable with a much greater magnitude of the rate of the `?duration` than that of 1 (e.g. `(inpocket ?p) += 1000 * ?duration`), so that the increase on the makespan (that is 0.0023 time units) would be insignificant. Exploiting this modelling trick would be acceptable for this example, but it would only be restricted to problems requiring a single control parameter per action. In this thesis, we have extended POPF by introducing the notion of control parameters, which cannot be mimicked using the duration parameter in problems with multiple non-discretised parameter setting.

The particular challenge we consider in this chapter, is how the value of the control

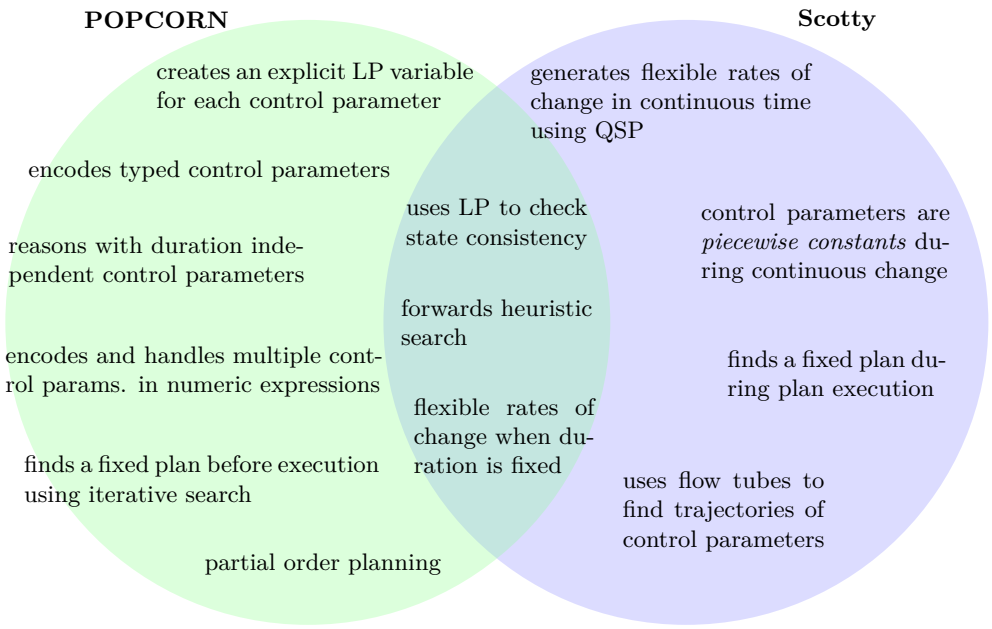


Figure 3.4: Comparison summary of POPCORN and Scotty systems.

parameter can be chosen by the planner, in particular when performing a forward state space search. In this example, the amount of cash we want to withdraw depends on which actions we apply after visiting the cashpoint. Early assignment of the value of a control parameter may lead to generation of poor plans (longer makespan). For instance, assigning a value to `?cash` before buying snacks would result in visiting the cashpoint twice. Therefore, the determination of the amount to withdraw should be made at a later stage in the plan construction (possibly after the entire action sequence has been constructed).

Our planner, POPCORN, builds up all the linear constraints acting on the control parameter `?cash` and (checking numeric consistency of every state visited using the linear program) until the goal state is reached. Then, the planner calls the final linear program to optimise all variables, e.g. `?cash`, subject to the metric objective of the problem. Since the metric objective is to minimise the `(inPocket ?p)` state variable in this example, the planner chooses the minimum bound of this variable as its value. We describe this process in detail in the remainder of the chapter.

3.3 Methodological Differences Between Scotty and POPCORN

Scotty is the planner that addresses the most similar class of problems to our system. The differences between the two regarding to modelling language, representations and methodology is summarised in Figure 3.4. We will discuss the differences in their LP encoding and the modelling languages they use in detail in further sections. In this section, we particularly focus on the differences in terms of how the value assignment is performed and and elaborate on the weaknesses of progressing in search using a fixed plan (recall that the Scotty planner finds a fixed plan before extracting a flexible plan).

Although the fixed plan generated by Scotty is a semantically meaningful symbolic plan (yet, temporally and numerically less-informative), it places the plan quality (e.g. the plan length) in peril. It ignores the fact that the early assignment of values for the control parameters (during search) has a direct impact on the number of times each action can occur in the plan. For instance, if the Scotty planner could solve the cashpoint example and chooses `?cash = 6` (that is in between the upper and lower bounds of the parameter) when finding a fixed plan, the plan would include four instances of `WithdrawCash` action whereas the goal can simply be achieved by a single execution of this action (as it is achieved by POPCORN). Although extracting a flexible plan removes this early assignment commitment, it does not help to find a shorter symbolic plan as it is not the concern of the QSP network. To fix this, the planner could make a smarter value assignment (i.e. `?cash` is assigned with its maximum value) that would possibly yield to a shorter solution plan, but deriving such an inference in a domain-independent way would be relatively challenging: not all numeric problems have similar characteristics. Although maximum value assignment could be a suitable inference on cashpoint domain, it would not work in problems requiring resource consumption (i.e. battery consumption problem in the simplified generator domain (Coles et al., 2009b)). This issue was not addressed by the authors of Scotty.

Another issue of using a fixed plan in Scotty is the following. When an action has two (or more) continuous effects with the same rate of change (i.e. two or more flow tubes that are inter-dependent to each other through the same control parameter), the value assigned to the rate may not be accommodating to both continuous changes at the same time. The authors acknowledges this issue with the following problem:

> "Imagine an activity *drive* with its control variable *speed*. Assume *drive* has two continuous effects that are modified by the control variable *speed*. On one hand the state variable x is modified by the flow tube $\Delta x = speed \cdot \Delta t$. On the other, the car's battery is drained according to $\Delta battery = -3 \cdot speed \cdot \Delta t$" (Fernández-González et al., 2015a, pg. 1569).

The problem in this example is that, when finding a fixed plan, the planner would prefer choosing a small value for the *speed* that consumes the battery and a large value that makes the car move fast. The observation makes it clear that a viable solution would be to avoid assigning a fixed value to *speed*, but in this case the constraints would become quadratic. The authors have recently tackled this issue by modelling the problem as Second Order Cone Programs (instead of modelling it as LPs), which allow them to solve such convex quadratic constraints (as mentioned in Section 2.6, the resulting system is called cqScotty (Fernández-González et al., 2017)). We will describe how cqScotty models these convex quadratic constraints in Section 3.4.1.

Recall that the Scotty and cqScotty planners do not support actions applying discrete numeric change. All numeric effects must be in the form of continuous change. In theory, both planners could express instantaneous change with a high rate of change and a tiny increment in time (e.g. $(t_{end} - t_{start}) = 0.001$) that approximates step functions. This would allow approximating the discrete numeric change as in the cashpoint problem. However, this would only work on one-dimensional variable update (as each continuous change supports only one control parameter in both planners), and consequently, the duration of the action

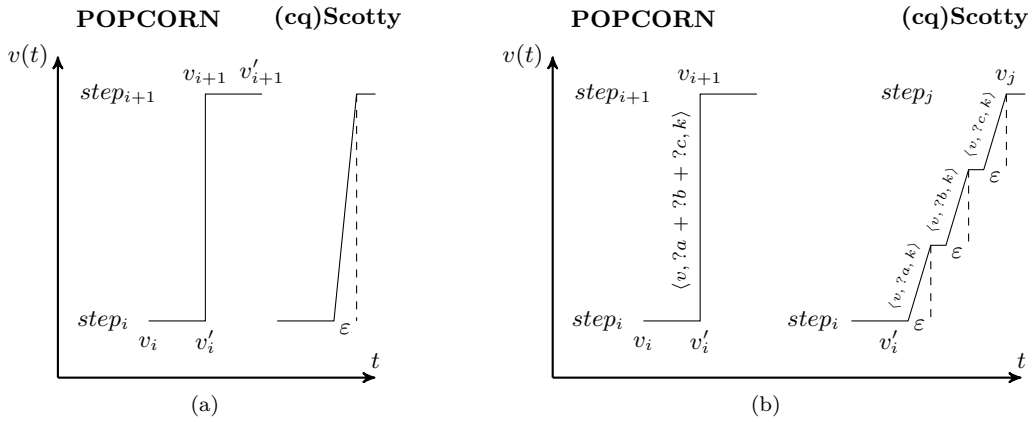


Figure 3.5: Discrete numeric change in POPCORN and a possible way to express it in Scotty and cqScotty. $\varepsilon = 0.001$ time units.

in the plan would be unrealistic (as the duration of 0.001 is highly inaccurate compared to the actual duration of the `WithdrawCash` action, which is 2). In case the intended discrete change is multi-dimensional (e.g. a state variable v is updated by two arbitrary control parameters $?a$, $?b$ and $?c$ at a step: $v \mathrel{+}= ?a + ?b + ?c$), the planner would create multiple additive continuous effects (with tiny increments in time) to mimic the discrete change. Figure 3.5 compares the discrete numeric change on variable v in POPCORN and how it could theoretically occur in Scotty. Sub-figures (a) and (b) illustrate one-dimensional and multi-dimensional (e.g. $v \mathrel{+}= ?a + ?b + ?c$) discrete changes, respectively.

3.4 Technical Background

As we have noted, duration can be seen as a special control parameter. POPF and its variants handle this parameter, so it forms a natural starting point from which to generalise to manage other control parameters. We therefore briefly review how the management of duration is achieved in POPF and COLIN.

A central innovation in POPF is the extended representation of a state. In classical forward search planners (discussed in Section 2.2.1), the states are valuations over the state variables (which may be boolean values in a propositional planner and boolean or numeric in a metric planner). PDDL2.1 planners must extend this representation to include a record of which durative actions are executing and when they started: we call this a *temporal state*. POPF further extends this representation, so that it searches in a space in which its states (P-states) represent *sets* of temporal states.

Definition 8 (P-state) *A P-state, S, is a tuple $\langle F, V, B, Q, P, C, T\rangle$, where:*

F is a partial valuation over boolean state variables.

V is a partial valuation over numeric state variables.

B maps the numeric state variables to upper and lower bounds (which might be infinite). Variables with a value defined in V have upper and lower bounds equal to this value.

Q is a set of actions which are started but not yet finished.

P is the collection of steps added to the plan to reach state S.

C is a collection of temporal and metric constraints accumulated over the steps in P.

T is the upper and lower bound on the time at which this state must end.

POPF only imposes a partial ordering on the actions in its plan (through temporal constraints included in C). POPF postpones commitment to ordering of actions when all orderings consistent with C are executable. This means that state variables whose values are not required to satisfy specific precondition constraints might be undetermined, subject to the resolution of the partial ordering of actions that affect them, which is why F and V are partial valuations. Where a continuous or duration-dependent effect is active in a P-state, the constraints in C tie together the bounds on the time in the current state (T) with the bounds on the value of the affected variable (B).

A P-state represents a set of temporal states because there can be many feasible solutions to C. The constraints in C can be purely temporal constraints (managed as an Simple Temporal Network) if there are no continuous or duration-dependent effects in the plan, or a linear program if time and numeric state variables interact (Coles et al., 2012). From C, values of B and T are found by solving C for upper and lower bounds on each variable.

POPF manages search through P-states by implementing both an appropriate P-state progression algorithm and also a variant of the relaxed plan heuristic, based on the Temporal Relaxed Planning Graph (see Section 2.6). The search choices faced by POPF are whether to add a new action start to extend a P-state, or else to complete an executing action (whose end is in Q). In both cases, constraints can be added to C to ensure preconditions and invariants are satisfied. In this way, POPF avoids branching on the infinite choice of values of duration, but manages constraints that restrict the possible values of durations, ensuring feasibility of the constraints at each P-state progression.

In order to implement temporal-numeric planning with control parameters POPCORN is built on the POPF planner. The partial-order mechanism in POPF minimises the addition of ordering constraints to avoid early commitment during forward search, in order to achieve flexible plans. The temporal constraints are added as they are needed to meet the preconditions of actions in a possible plan. The existing partial-order mechanism of POPF helps POPCORN to avoid early-commitment in assigning values to the control parameters. As discussed in Section 3.2, the early commitment in the valuation may lead to poor plans. The constraints implied by the pre- and post-conditions of actions added to the plan govern the possible values of control parameters (i.e. restricts the feasible region), as illustrated in Figure 3.6. The details of this process are described in the rest of this chapter.

3.4.1 LP Temporal and Numeric Scheduling

The POPF planner inherits the use of linear programming from the COLIN planner. It uses the LP to check the temporal and the numeric consistency of a P-state. The P-state variables that capture evolution of the numeric state variables along the trajectory of a plan are defined as follows:

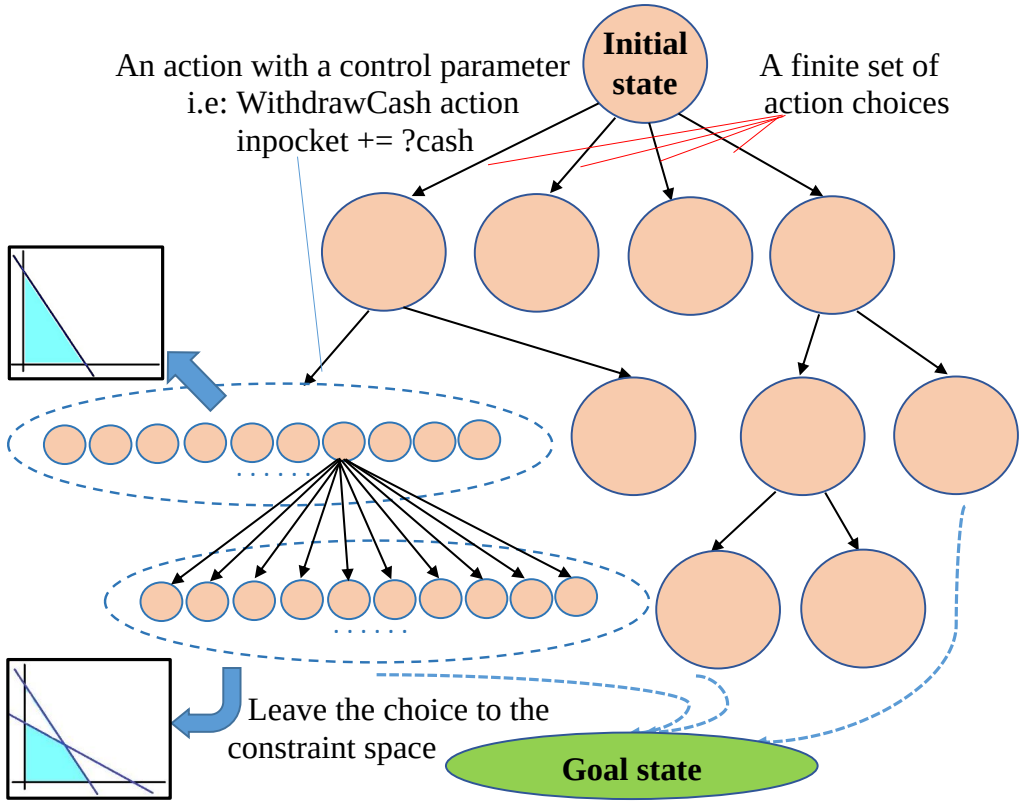


Figure 3.6: Schematic representation of the search space where there is a control parameter effect. The nodes represent the state reached, and the edges represent the action applied to reach the next state. The graphs in black boxes represent the LP constraint space, which is used to avoid complex branching choice.

A sequence of pairs of vectors of variables, V_i and V_i', is constructed, one for each step i in the plan. Each $v_i \in V_i$ records the value of the state variable v *just before* the step i. Similarly, each $v_i' \in V_i'$ records the value of a state variable v *immediately after* the step i. For instance, a state variable v can have a discrete instantaneous numeric change at a step i. In this case, the constraint $v_i' = v_i + c$, where v_i is increased by the constant value of c at the step i, is added to the LP to record this change. The numeric change (including the continuous linear change) in COLIN is discussed in detail in Section 2.6. The numeric change in POPCORN occurs using the same formalism of POPF (as described above). However, since the POPCORN planner is restricted to linear numeric change, the continuous numeric change with control parameters is not yet explored. Based on this representation, Table 3.1 shows the constraints and variables created to record numeric changes that are connected to the $?cash_{(i,s)}$ control parameter, where i and s denote the unique action and step indices, respectively. $balance_s$ and $inpocket_s$ represents the $(balance\ ?m)$ and $(inPocket\ ?p)$ state variables at step s, respectively. $[lb,\ ub]$ denotes the lower and upper bounds of the variables at a state. The observation makes clear that each numeric constraint imposed can constrain the bounds of the variables (and the control parameters).

The rationale behind encoding each control parameter with unique action and step indices in the LP is the following. Firstly, as mentioned earlier, the scope of each control parameter

Table 3.1: Variables and constraints acting upon the `?cash` parameter, that are collected from the initial state to reach the goal state.

Plan Action	LP Variable	[lb, ub]	Constraints
Withdraw	$?cash_{(0,0)}$	[5, ∞]	$?cash_{(0,0)} \geq 5$
(start)		[0,50]	$?cash_{(0,0)} \leq bal_0$
i = 0	bal_0	[50,50]	$= bal_0$
	bal'_0	[0,45]	$= bal_0 - ?cash_{(0,0)}$
Withdraw	$?cash_{(0,1)}$	[5,50]	$= ?cash_{(0,0)}$
(end)	inp_1	[2,2]	$= inp_0$
	inp'_1	[7,52]	$= inp_1 + ?cash_{(0,1)}$
BuySnacks	inp_2		$= inp'_1$
(start)	inp'_2	[7,52]	≥ 5
BuySnacks	inp_3	[7,52]	$= inp'_2$
(end)	inp'_3	[2,47]	$= inp_3 - 5$
Go Pub	inp_4	[2,47]	$= inp'_3$
(start)	inp'_4	[2,47]	$= inp_4$
Go Pub	inp_5	[2,47]	$= inp'_4$
(end)	inp'_5	[20,47]	≥ 20

is limited to the action, in which it is defined, so that its LP variable must be unique to the action. Secondly, it is possible to have repeated occurrences of an action in the plan, however in the implementation of POPF and its predecessors, each occurrence can only appear once at a step. Thus, using both the action and the step indices as its key values prevents re-defining previously defined variables in the LP. We provide a generalised formalisation of encoding the control parameters in the LP in Section 3.5.4.

The Scotty planner also inherits the use of LP from COLIN. In contrast to our approach, they avoid defining an LP variable for each control parameter that are encountered along the trajectory of states in order to avoid non-linear interaction. Instead, each control parameter is assigned with a *piecewise constant* between consecutive steps. Recall that the Scotty planner updates the value of a state variable x with a single control parameter d in continuous effect between consecutive steps i and j: $x(j) = x(i) + k \cdot d \cdot (t_j - t_i)$, where t_j and t_i are times of steps j and i, respectively, k is a constant. The control parameter d is assigned with a constant, say $d_{i \to j}$, within the LP and it is computed as follows:

$$d_{i \to j} = \frac{x(j) - x(i)}{k \cdot (t_j - t_i)} \tag{3.1}$$

To illustrate the differences of LP encoding of control parameters in both systems, consider the following abstract example. Suppose that a trajectory of LP steps are generated by applying various actions, where each updates an arbitrary state variable with a single control parameter, yields a final step $step_m$ (starting from $step_k$, where $k < m$) that needs to be checked for consistency by the LP. The values of the control parameters between these steps encoded in LP by Scotty and POPCORN are shown in Figure 3.7. Observe that the control parameters in POPCORN are encoded as flexible LP variables (bounded their upper and lower bounds), whereas in Scotty they are assigned with piecewise constants. The observation makes clear that the LP generated by Scotty only checks the numeric consistency

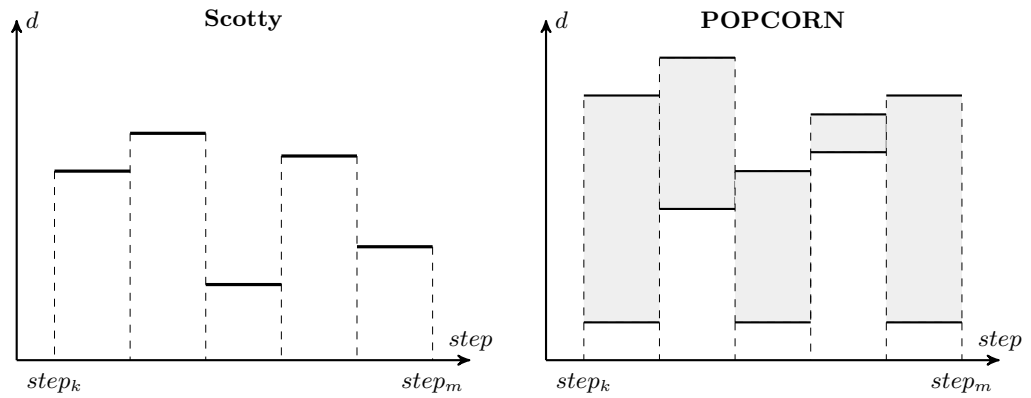


Figure 3.7: Control parameter LP variable values between and at steps of both systems.

of the step $step_m$ with fixed values of control parameters ignoring other possible values the parameters can take. This process yields finding a fixed solution plan that is highly fragile to the changes in the environment. As discussed in Section 3.3, the Scotty planner finds a temporally and numerically flexible plan using QSP once a fixed solution plan is found. Then, we can say the temporal-numeric consistency of a flexible plan generated by Scotty is checked by the QSP network (*after* a totally ordered fixed plan is reached), not by the LP inherited from COLIN.

The cqScotty planner avoids using piecewise constants for the values of the control parameters in continuous effects (but they stay constant in numeric preconditions to simplify the model, as in Scotty), when modelling Second Order Cone Programs (SOCP) to check temporal-numeric consistency of a state. Although they do not explicitly define decision variables for control parameters in SOCP, they define a new auxiliary variable $c_{i\to j}$ that represents the product of the control parameter d and the time between two consecutive steps, t_j and t_i, where $j = i + 1$ and this representation is formulated as follows:

$$c_{i\to j} = d \cdot (t_j - t_i) \tag{3.2}$$

This variable is constrained by the linear constraints of the product of the upper and lower bounds of d (i.e. $ub(d)$ and $lb(d)$, respectively) and the time between steps i and j, and formulated as:

$$lb(d) \cdot (t_j - t_i) \leq c_{i\to j} \leq ub(d) \cdot (t_j - t_i) \tag{3.3}$$

Using this representation, the non-linear formulation of $x(j) = x(i) + k \cdot d \cdot (t_j - t_i)$ becomes linear: $x(j) = x(i) + k \cdot c_{i\to j}$.

Additionally, the cqScotty planner can constrain the norm (or the magnitude) of a vector of control parameters using convex quadratic constraints. For instance, let vel_x and vel_y be two control parameters denoting the velocities of a robot in x- and y-directions and $\mathbf{d}$ be a vector of these control parameters: $\mathbf{d} = [vel_x, vel_y]$. The l_2-norm constraint on $\mathbf{d}$ with a constant vel_{max} is $\|\mathbf{d}\| = vel_x^2 + vel_y^2 \leq vel_{max}^2$. This equation is multiplied by $(t_j - t_i)$ on both sides to obtain the SOCP constraint $\|\mathbf{d}\| \cdot (t_j - t_i) = \|\mathbf{c_{i\to j}}\| \leq vel_{max} \cdot (t_j - t_i)$. The same trick is used in the continuous decrease effects to overcome the weaknesses of finding

fixed plans (discussed in Section 3.3).

3.5 Planning with Control Parameters

The difference between the POPF and POPCORN planners is that POPCORN can reason with control parameters in addition to duration. Control parameters are, as with duration, variables that appear only as part of the structure of an action in the plan, not as part of the state. This means that variables can be added to the constraint collection in a P-state as new actions are applied, to represent not only the values of the state variables, but also newly introduced control parameters. This is achieved by extending the existing machinery in POPF. We consider the details of each component in their related subsections. We extend the existing problem and state definitions to capture the control parameters defined in actions. We define the additional constraints and variables added to the LP based on the numeric preconditions and effects of actions added during state progression. We provide details of the modifications made to the existing heuristic state evaluation of POPF and analyse the effects of the control parameters on the search space. We use the cashpoint example to illustrate elements discussed in the related subsections.

3.5.1 Problem Definition

In this section we present extending temporal and numeric planning formalism discussed in Section 2.2.2 and Section 2.2.3 to include control parameters. The definition respects the full semantics of the PDDL2.1 language and additionally includes control parameters.

Let $\mathbf{v}$ be a finite set of numeric variables (of size r) and $\mathbf{d^a}$ be a finite set of control parameters (of size m^a) defined in an action $a \in A$. Each numeric variable (e.g. `(inPocket Joe)` and `(balance ATM1)` in the cashpoint example) is denoted by $v \in \mathbf{v}$. The i^{th} control parameter of the action a is denoted by $d_i^a \in \mathbf{d^a}$. The `?cash` parameter is the only control parameter of the `WithdrawCash` action, thus the size of its control parameter set is 1 (i.e. $m^{WithdrawCash} = 1$).

Let $f(\mathbf{v}, \mathbf{d^a})$ be an arbitrary function applied to $\mathbf{v}$ and $\mathbf{d^a}$. Let $g(\mathbf{v}, \mathbf{d^a}, \texttt{?duration})$ be an arbitrary function applied to $\mathbf{v}$, $\mathbf{d^a}$ and a special control parameter, `?duration`, representing the duration of the action in which it is defined. Moreover, let $comp \in \{\geq, >, \leq, <\}$, $op \in \{\nearrow, \leftarrow, \searrow\}$ be sets of comparison and assignment operators, respectively, where $\nearrow$ denotes increase assignment, $\leftarrow$ denotes direct assignment and $\searrow$ denotes decrease assignment operators.

We define the temporal and numeric planning problem with control parameters as follows:

Definition 9 (Temporal-Numeric Planning Problem with Control Parameters) *A temporal and numeric planning problem with control parameters is a tuple* $\langle F, \mathbf{v}, I, A, G, M \rangle$*, where:*

- F *is a finite set of grounded literals,*
- $\mathbf{v}$ *is a finite set of numeric variables,*
- I *is the initial state,*

- *A is a set of actions. Each action, $a \in A$, is a tuple:*

$$a = \langle dur,\ cont,\ pre^n_x,\ pre_x,\ eff_y{}^n,\ eff_y \rangle$$

  - *dur is a set of duration constraints of action a, where each is expressible in the form of $\langle g(\mathbf{v}, \mathbf{d^a}, \texttt{?duration}), comp, c\rangle$, where $c \in \mathbb{R}$.*
  - *cont is a set of constant bounded control parameter constraints of action a in the form of $\langle f(\emptyset, \mathbf{d^a}), comp, c\rangle$.*
  - *pre_x and $pre^n{}_x$ are the propositional and numeric conditions, respectively, that must hold at a state in which the action starts ($\vdash$), at a state in which the action ends ($\dashv$), or between the start and the end of an action a ($\leftrightarrow$): $x \in \{\vdash, \leftrightarrow, \dashv\}$. The numeric conditions are in the form:*

$$pre^n_x = \langle f(\mathbf{v}, \mathbf{d^a}), comp, c\rangle$$

  - *eff_y are the propositional start (or end) effects of an action a that are to be added to (or deleted from) the world state. $eff_y{}^n$ are the numeric effects that act on variable $v \in \mathbf{v}$. We denote the decrease effects by $eff_y{}^-$ and the increase effects by $eff_y{}^+$; where $y \in \{\vdash, \dashv\}$. The numeric effect $eff_y{}^n$ is expressible in the form:*

$$eff_y{}^n = \langle v, op, g(\mathbf{v}, \mathbf{d^a}, \texttt{?duration})\rangle,$$

- *G is the goal described by a set of propositions, $p(G) \subseteq F$, and a set of numeric conditions, $v(G)$, over the state variables,*
- *M is the metric objective function.*

A constant bounded control parameter constraint (in the form of $\langle f(\emptyset, \mathbf{d^a}), comp, c\rangle$) refers to a numeric precondition that can limit the bounds of control parameter(s) with a constant number (e.g. in cashpoint example we have $\texttt{?cash} \geq 5$). On the other hand, a non-constant bounded control parameter constraint (in the form of $\langle f(\mathbf{v}, \mathbf{d^a}), comp, c\rangle$) refers to a numeric precondition that can limit the bounds of control parameter(s) with state variable(s) (e.g. $\texttt{?cash} \leq (balance?m)$), which means their values are inter-dependent on each other. We see that these constraints are recorded separately in the definition. The reason behind this is that constant bounded control parameter constraints are almost negligible during the heuristic computation, and they are directly sent to the LP. However, the non-constant ones require further investigation in the heuristic. Overall, the separation aims to decrease the complexity of the heuristic that we will introduce in Section 4.

A solution to a temporal-numeric problem with control parameters is an extended version of the temporal-numeric plan described in Section 2.2.3:

Definition 10 (Controlled Temporal-Numeric Plan (π)) *is a list of timestamped actions with duration and flexible values of the control parameters of actions in π. It is shown as:*

$$\pi = \{\langle a_i, ts_i, \Delta t_i, \mathbf{D^{a_i}}\rangle \mid \forall i = \{0, \ldots, n\}\},$$

where n is the plan length, $ts_i \in \mathbb{R}$ is a timestamp (the time at which the action a_i is applied), $\Delta t_i \in \mathbb{R}$ is the duration and $\mathbf{D^{a_i}}$ is an $(m^{a_i} \times 2)$ matrix (of rational numbers, i.e. $\mathbf{D^{a_i}} \in \mathbb{R}^{(m^{a_i} \times 2)}$) of the lower and the upper bound values of the control parameters of a_i. The plan generates a sequence of at most $2 \cdot n$ states (each is denoted by $s(t)$) by applying the start and the end of action a_i at time-points $(t = ts_i)$ and $(t = ts_i + \Delta t_i)$, respectively. Additionally, the following conditions must hold true:

- *the duration constraints, dur, of each action in π are satisfied,*
- *the constant bounded control parameter constraints, cont, of each action in π are satisfied,*
- *$pre_{\vdash}(a_i)$ and $pre_{\dashv}(a_i)$ hold in states $s(ts_i)$ and $s(ts_i + \Delta t_i)$, respectively, such that $\forall i = \{0, \ldots, n\}$,*
- *$pre_{\leftrightarrow}(a_i)$ hold in every state $s(t)$, where $ts_i < t < ts_i + \Delta t_i$, such that $\forall i = \{0, \ldots, n\}$,*
- *The trajectory of states generated by application of the actions yields a final state that satisfies the goal.*

Observe that, in Definition 10, the solution does not specify a fixed value to each control parameter. Instead, it reports the lower and upper bound of possible values that each control parameter can take. We will present another solution to the problem, in which their values are iteratively chosen by an executive in Section 3.5.5.1. Table 3.2 shows a possible (and a desired) controlled plan to our running example.

Table 3.2: A desired controlled plan for the cashpoint example.

Timestamp	**Action**	**Duration**	**Control Parameters**
0.001	(Go Joe Home Bank)	5	
5.002	(WithdrawCash Joe Bank ATM1)	2	?cash::[23, 50]
7.003	(Go Joe Bank Store)	5	
12.004	(BuySnacks Joe Store)	1	
13.005	(Go Joe Store Pub)	5	

3.5.2 State Definition

Having presented the conceptual model of the planning task, we investigate how the task is represented in the state space. The P-state representation used in POPF is further extended for temporal-numeric planning with control parameters. We call this extended representation a C-state (Control state).

Definition 11 (C-state) *A C-state, S, is a tuple $\langle F, V, B, Q, P, C, T, D \rangle$, where: F, V, B, Q, P and T are as defined in a P-state. D maps all control parameters associated with actions in P to pairs recording the upper and lower bounds on their values in S.*

When a new action, A, is started, the state progression for POPCORN adds to D newly created variables corresponding to the control parameters in A, with upper and lower bounds derived from the preconditions of A (or $+\infty$ or -∞ where no bounds appear in A). Constraints are added to C that represent the pre- and post-condition requirements on these control parameters.

State progression of C-states is similar to P-state progression: new actions can be started, or current actions can be completed. In both cases, the appropriate constraints are added to C, introducing control parameters where necessary and new state variables, and new bound are then computed for B, T and D.

```
(:action descend
 :duration (d)
 :precondition (and (y <= 200))
 :effect ()
 :dynamics (4 <= vx <= 8, 3 <= vy <= 6))
```

Figure 3.8: The Kongming action model for AUV descent. Note that this is not PDDL, but a variant in which a new `:duration` field is added to classical actions to associate a fixed and equal duration to *all* actions, and `:dynamics` which defines the control parameters and their bounds.

3.5.3 Modelling Actions with Control Parameters

In our proposed language extension to PDDL2.1, the ordering of the duration and control fields in action descriptions is carefully chosen to allow control parameters to appear in duration constraints. This makes it possible to tie together these parameters in constraints. For example, we can capture a more general version of the Kongming AUV descend action, shown in Figure 3.8, allowing a variable duration descent that is also able to descend at different gradients by separately selecting the x- and y-velocities. The assumption modelled in the action is that the execution of this action involves moving at a constant velocity along a straight vector with the chosen x- and y- displacements. To avoid the non-linear interaction between rates and durations, we model this by choosing the distances travelled in the x- and y-directions, subject to the constraints that these distances must not imply violation of the velocity limits. We can also combine multiple control parameters in a single constraint, as illustrated in the `descend` action in Figure 3.9. This constraint ensures that the combined power consumption of the motors (i.e. the power consumption of motors that traverses the vehicle in the x- and y- directions: `?dx * (powerX ?a) + ?dy * (powerY ?a)`) does not exceed the available power output (i.e. `?duration * (power ?a)`). This constraint is an invariant condition ensuring that it holds throuhgout the execution of the action. Having multiple control parameters makes the control space for this problem multi-dimensional. This constraint is in the form of $\langle f(\mathbf{v}, \mathbf{d^a}), comp, c\rangle$. Also note that there are 4 duration constraints (e.g. `(<= ?duration (/ ?dx (minVx ?a)))`) in the `descend` action schema, where each are expressible in the form of $\langle g(\mathbf{v}, \mathbf{d^a}, \texttt{?duration}), comp, c\rangle$. This set of duration constraints ensure that the duration of the action lies within the bounds that are restricted by the varying displacement on each direction with the predefined maximum

and minimum velocity rates. As the duration and power usage constraints hold, the action increases the position of the vehicle `?a` by the displacement rates, `?dx` and `?dy`, at the end of its execution.

```
(:durative-action descend
 :parameters (?a - auv)
 :control (?dx ?dy - number)
 :duration (and
  (<= ?duration (/ ?dx (minVx ?a)))
  (<= ?duration (/ ?dy (minVy ?a)))
  (>= ?duration (/ ?dx (maxVx ?a)))
  (>= ?duration (/ ?dy (maxVy ?a))))
 :condition (and
  (over all (<= (+ (* (powerX ?a) ?dx) (* (powerY ?a) ?dy))
            (* ?duration (power ?a)))))
 :effect (and   (at end (increase (posX ?a) ?dx))
                (at end (increase (posY ?a) ?dy))))
```

Figure 3.9: Descend action with control parameters. This model is in PDDL2.1 extended with our proposed syntax for control parameters. Note that the linear power constraint (as a condition) restricts the total power use across the two motors according to the consumption rates of the motors.

Scotty (Fernández-González et al., 2015a) offers a syntax for representing control parameters that is very similar to our own. In Scotty, control parameters are intended to represent the margins of control of processes during execution, so these parameters are always intended to be interpreted as rates, as can be seen in their use in the continuous effects in Figure 3.10. However, this model combines duration and control parameters in a quadratic expression. Scotty overcomes this difficulty by focussing on upper and lower bounds defining the range of possible values for the control parameters. This use of control parameters cannot be exploited in problems such as the cashpoint domain, where the parameter is not a rate.

```
(:durative-action navigate
 :control-variables ((velX) (velY))
 :duration (and (<= ?duration 5000))
 :condition (and
  (over all (>= (velX) -4)) (over all (<= (velX) 4))
  (over all (>= (velY) -4)) (over all (<= (velY) 4))
  (over all (<= (x) 700)) (over all (>= (x) 0))
  (over all (<= (y) 700)) (over all (>= (y) 0))
 :effect (and
  (increase (x) (* 1.0 (velX) #t))
  (increase (y) (* 1.0 (velY) #t))))
```

Figure 3.10: The `navigate` action for Scotty (Fernández-González et al., 2015a). The syntax is very similar to our proposal, but the control parameters (e.g. velX and velY) are always rates of change.

3.5.4 Checking C-state Consistency

It is worth emphasising the main characteristic of control parameters within an action instance: each control parameter is a *local* variable, whose scope is limited to the action

within which it is defined. They are carried out through the plan with the help of *global* variables (e.g. *(inPocket ?p)* in the cashpoint domain). Each control parameter is introduced in constraints only at the point when an action is applied. Subsequent constraints can impact on the values of control values by constraining variables that also appear in constraints with the control parameters.

Having presented an example encoding of the control parameter constraints in POPCORN, within the LP (using the constraints in C) in Section 3.4.1, we now present a formal structure on the linear constraints. Let a be the index of a grounded action appearing in step s with m^a control parameters. The index of each control parameter is j, where $j = \{0, \ldots, m^a - 1\}$. Suppose that $\mathbf{v}$ is the vector of state variables (as they appear in the LP), $\mathbf{w}_v$ is a vector of constant coefficients of the LP state variables $\mathbf{v}$, $\mathbf{d^a}$ is a set of control parameter LP variables defined in action a and each can be shown as $d(j)_{(a,s)}$, $\mathbf{w}_d$ is a vector of constant coefficients of $\mathbf{d^a}$, c is a constant and $f(\mathbf{v}, \mathbf{d^a})$ and $f(\emptyset, \mathbf{d^a})$ are linear functions applied to r numeric variables ($\mathbf{v}$), and m^a control parameters ($\mathbf{d^a}$). The functions can be shown as:

$$f(\mathbf{v}, \mathbf{d^a}) = \mathbf{w}_v \cdot \mathbf{v} + \mathbf{w}_d \cdot \mathbf{d^a} + c$$

$$f(\emptyset, \mathbf{d^a}) = \mathbf{w}_d \cdot \mathbf{d^a} + c$$

The *comp* and *op* are the comparison and assignment operators as defined earlier.

In addition to the existing temporal and numeric constraints that can appear in POPF, our approach inserts the following constraints to the LP during C-state progression:

- Any numeric precondition that is given in the form:

  $\langle f(\mathbf{v}, \mathbf{d^a}), comp, c\rangle$ (non-constant bounded control parameter constraint)

  $\langle f(\emptyset, \mathbf{d^a}), comp, c\rangle$ (constant bounded control parameter constraint)

- Any numeric effect that is in the form:

  $$\langle v, op, f(\mathbf{v}, \mathbf{d^a})\rangle$$

  $$\langle v, op, f(\emptyset, \mathbf{d^a})\rangle$$

The main difference of the constraint formulations encoded in the LP and task (given in Definition 9) formalism is that the control parameter is defined as a global variable in the LP using its unique identifiers (e.g. unique action, step and control parameter indices). In contrast, in the planning task definition, it is defined as a local action variable. This plays a critical role when propagating constraints with such local parameters in C-state progression.

Following extension of C as part of C-state progression, C is tested for feasibility. If it is infeasible, then the C-state corresponds to an empty set of temporal states, so it is pruned and search reverts to an earlier C-state. In POPCORN, C is solved for minimum and maximum bounds T, B and D. However, an alternative possibility is to tighten bounds selectively. Furthermore, a solution to C can act as a *witness* that can be carried forward in the C-state, rather like a watched literal in modern SAT-solvers, checking it against new constraints as they are added and only resolving C when the current witness fails.

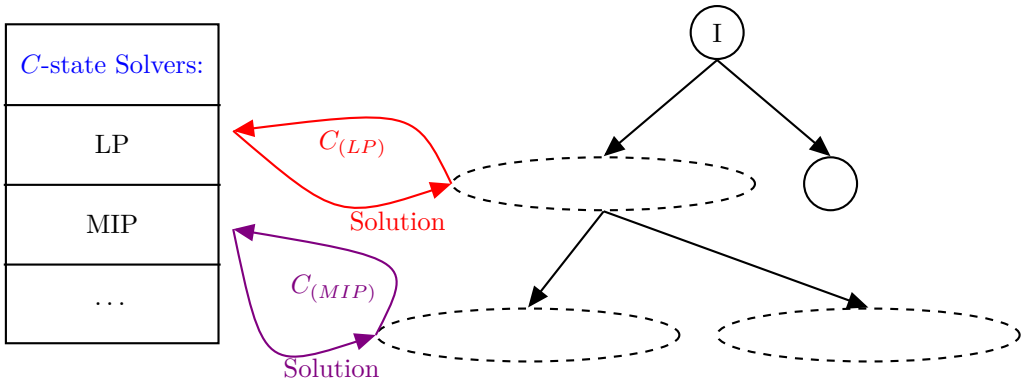


Figure 3.11: Illustration of state consistency checking according to type of the accumulated constraints.

The requirement for state checking is that it is possible to check C for feasibility: this does not necessarily require C to be a linear program. In principle, we can use mixed collections of constraints, possibly separating C into disconnected sets of constrained variables, using constraint solvers appropriate to each set of constraints according to type. As a simple example, we experiment with integer control variables, making C a mixed integer program (MIP). We envisage the possibility to extend this idea to different types, using specialised constraint solvers to handle them (see Figure 3.11).

3.5.5 Temporal and Numeric State-Space Search

The duration of an action might not be fixed. It might be determined by either the values of metric fluents, such as `(<= ?duration (v ?p))`, or constrained within a range of values (Coles et al., 2009b), for example, `(and (>= ?duration 10) (< ?duration 50))`. Likewise, the value of a control parameter defined in an action is usually not fixed, but is constrained within an interval and, possibly, constrained by multiple constraints. The values of state variables subject to continuous or duration dependent effects are similarly constrained within ranges, $[v^{min}, v^{max}]$. The planner is free to choose the value of a variable within its bounds. The choices might not be independent, so each choice must be propagated through C to determine how bounds of other variables are affected.

The existence of control parameters generates a complex branching choice in the search space, just as is the case for `?duration` parameters, but possibly multi-dimensional. Figure 3.6 illustrates this effect for our cashpoint example. When the planner branches over this set of infinitely many states, it avoids exploring each state by leaving the choice to the *LP constraint space*.

Although the solver can help the planner to prune numerous possible states in a C-state (in other words, it helps to shrink the size of the C-state), the intervals (of control parameter) are usually not tight enough to easily draw a decision on their values. In our system, we leave the valuation of parameters until a partially-grounded plan is reached (a solution is found) and have accumulated a collection of constraints on the way. Each control parameter has an interval of feasible values; otherwise the reached plan is numerically infeasible. At this point, there are two options to proceed on the valuation. The first option is that

we can define a metric objective function to optimise a linear function (e.g. in cashpoint example: `(:metric minimize (inPocket Joe))`), so that the planner optimises the control parameter in sequence (or possibly in a desired order of importance). The optimality is subject to the metric objective function defined. We presented an example on this method in Section 3.2. The second option is to leave the decision to the executive. We call this method as *controlled plan management* and provide the details in the next subsection.

3.5.5.1 Managing a Controlled Plan

As mentioned in Section 3.3, the Scotty planner provides a flexible plan to an executive and the executive gets freedom to choose any value to each control parameter within its bounds. The authors of the work of Scotty ignore the fact that the bounds of control parameters can be inter-dependent to each other (i.e. a value decision drawn by the executive can potentially change the feasible bounds of other parameters). This issue slightly complicates the assignment process and requires continuous revision of the feasible bounds of these parameters. In this section, we tackle this issue by iteratively optimising each parameter subject to the temporal-numeric constraints that are propagated through the trajectory of states that yields a flexible plan (or a controlled plan π, see Definition 10). This process can be considered as a post-processing step, and in principle, it is independent from the plan execution. This contrasts to the approach of Scotty, where the process is undertaken during plan execution. We introduce the general approach in this section and consider a scenario, in which the process occurs in execution time (as in Scotty) in Section 5.3.2.

```
(:durative-action GiveMoney
 :parameters (?p - person ?a - location)
 :control (?change - number)
 :duration (= ?duration 1)
 :condition (and (at start (at ?p ?a)) (over all (at ?p ?a))
  (at start (beggarAt ?a))
  (at start (>= ?change 1))
  (at start (<= ?change (inpocket ?p)))
 :effect (and
  (at end (moneyGiven))
  (at start (decrease (inpocket ?p) ?change))))
```

Figure 3.12: The `GiveMoney` action of the extended cashpoint example.

Let S_{FINAL} be the goal state reached at the end of the forward state-space search and C_{FINAL} be the final collection of temporal-numeric constraints accumulated until reaching the state S_{FINAL}. At the end of planning, POPCORN returns a controlled plan π that consists of orderings of timestamped activities and flexible intervals of their control parameters (if they exist). Choosing a value within any of these intervals can affect the following attributes of the controlled plan:

1. The timestamps and the durations of actions. This occurs if the duration parameters of actions are linearly dependent on the control parameter, whose value is recently assigned. Consequently, the orderings of actions can change.

2. The interval bounds of the other control parameters.

Remark 1 *Understandably, one can be concerned about the effects of this process on the timestamps, the durations and the orderings of actions in π. Recall that C_{FINAL} includes decision variables for the timestamps of actions as well as for their control parameters. It also includes temporal ordering constraints between actions and duration constraints of actions in π. Therefore, any numeric decision enforced to C_{FINAL} can easily update these attributes of the plan, including the bounds of other control parameters.*

The choices made for each parameter must be propagated through the constraint space (starting from the C_{FINAL}), so that the control parameter bounds (as well as other attributes of the plan) can be updated iteratively. This process is indeed a bounding search, in which state propagation decisions are undertaken by an executive agent (e.g. a human operator) or by a control system (e.g. a motion planner). We refer to this post-processing approach as *controlled plan management*. In this section, we investigate the approach in a scenario where the decision-maker is a human operator.

Consider an extended version of the cashpoint example (presented in Section 3.2), where we want to have given some cash to a beggar on the way to the pub. The amount of cash we can give is modelled as a control parameter (`?change`), whose bounds are within [1, `(inPocket ?p)`] in sterling (see Figure 3.12). A possible controlled plan generated for this problem is given in Table 3.3. The `WithdrawCash` and `GiveMoney` actions consist of local control parameters (e.g. `?cash` and `?change`), whose values are inter-dependent of each other through the state variable `(inPocket ?p)`. Thus, the valuation of any of the two control parameters affects the other. For instance, if the executive decides to withdraw £30 (that is in the feasible bounds of `?cash`), then the maximum value `?change` can take will decrease from £27 to £7[1].

It is also possible that the executive can choose a value for the `?change` parameter (say it is £10) before deciding on how much to withdraw. In this case, the minimum value `?cash` can take will increase from £24 to £33. The ordering of the value assignment is critical. Having already assigned `?change` = £10, the assignment of £30 to `?cash` will no longer be a feasible assignment.

Table 3.3: A plan generated by POPCORN for the extended cashpoint example.

Time	**Action**	**Duration**	**Control Parameters**
0.001	(Go Joe Home Bank)	5	
5.002	(WithdrawCash Joe Bank ATM1)	2	?cash::[24, 50]
7.003	(Go Joe Bank beggar)	5	
12.004	(GiveMoney Joe beggar)	1	?change::[1, 27]
13.005	(Go Joe beggar Store)	5	
18.006	(buySnacks Joe Store)	1	
19.007	(Go Joe Store Pub)	5	

It is somewhat easy to compute the values of updated bounds for this example, but the computation becomes relatively difficult when the complexity of the problem increases.

[1] A simple LP model is constructed and solved manually to find the values. The model can be found at https://github.com/Emresav/ECAI16Domains/blob/master/beggarModel

Thus, we propose solving C_{FINAL} (plus, the value assignment constraints) iteratively using a mathematical solver (e.g. LP or MIP) until we reach a *bound* (until all control parameters are assigned to a fixed value).

Controlled plan management can be considered as a search problem whose nodes represent mathematical models (it is either a linear or a mixed integer model) consisting of temporal-numeric constraints (denoted as E) and edges represent the state transitions based on executive decisions over the control parameters (denoted as r). We refer to each node E as *a model state*, since it forms a mathematical model to be solved. The initial state of the search is equivalent to C_{FINAL}. Although the process is modelled as a search problem, it does not adopt any state space search methods (e.g. breadth-first). Instead, the branching scheme is subjective to the executive: it is based on the sequence of decisions drawn by the executive. If an infeasible state is reached, the executive has the power to backtrack (or backjump) at any state model that is recorded to be feasible in the search space. Figure 3.13 shows the iterative controlled plan management search for this example.

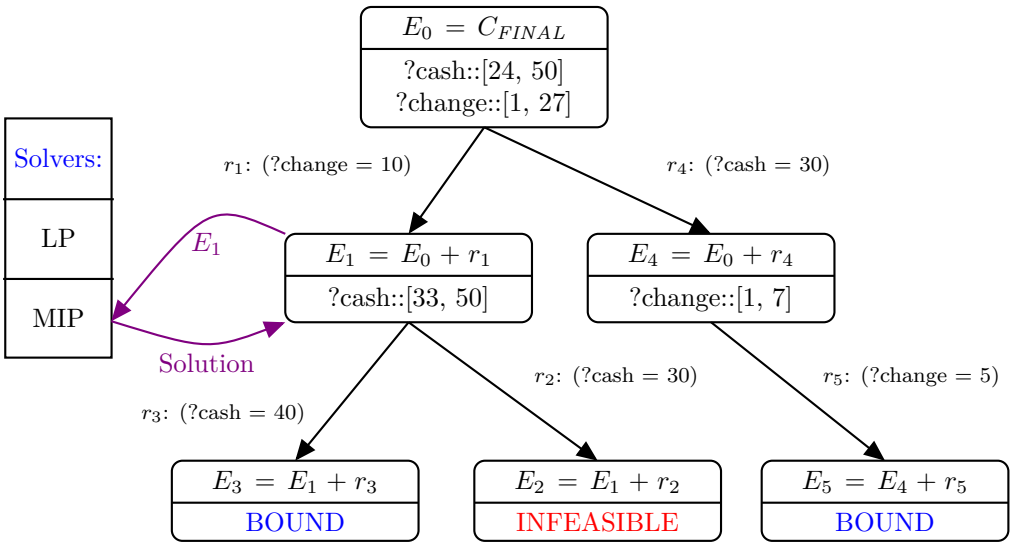


Figure 3.13: Schematic representation of the controlled plan management for the extended cashpoint example.

In principle, the controlled plan management can be implemented with Algorithm 1. The algorithm takes C_{FINAL}, uses it as the initial state and at the end it returns a list of executive value decisions of each flexible control parameter (denoted as $R(E)$). It propagates all the temporal-numeric constraints (i.e. the accumulated assignment decisions each denoted as r, and the initial constraint set C_{FINAL}) through model states until at a model state E, in which each control parameter is assigned with a fixed value (i.e. $\mathbb{D}(E) = 0$). Then, the algorithm reports a list of control parameter assignments as a solution. The controlled plan management constructs the theoretical grounds of managing partially grounded plans before or during execution. Since the plan execution is not in the scope of this thesis, we have not yet implemented this feature on our framework. We will combine POPCORN with the ROSPlan (Cashmore et al., 2015), a framework that integrates domain-independent planners and execution tools on ROS (Quigley et al., 2009), in the future. Further details of this work is outlined in Section 6.2.3.

Note that the search graph only expands on model states that are deemed to have feasible solutions. The search traverses purely based on model state choices drawn by the executive (from the list $\mathcal{E}$). The termination criteria on the algorithm is rather open-ended. It only terminates when there is no longer flexible control parameters in a model state, which purely relies on how efficient the executive is during the decision-making process. Overall, the executive is the only mechanism who makes state propagation (and termination) choices on feasible regions modelled by the algorithm.

Algorithm 1: Controlled plan management

Data : C_{FINAL}
Result: $R(E)$ - a list of executive assignments to reach the model state E
$\mathcal{E}$ - list of feasible model states, r - an executive assignment decision at a model state;
$\mathbb{D}(E)$ - a list of flexible control parameters and their bounds in E;
$R(C_{FINAL}) \leftarrow \emptyset$, $\mathcal{E} \leftarrow C_{FINAL}$;
Dispatch $\mathcal{E}$ to the executive and ask to choose a model state E;
while $\mathbb{D}(E) \neq \emptyset$ **do**
    Dispatch $\mathbb{D}(E)$ to the executive;
    Retrieve r from the executive;
    if $Solve(E + r)$ *succeeds* **then**
        $E' \leftarrow E + r$;
        $R(E') \leftarrow R(E) + r$;
        $\mathbb{D}(E') \leftarrow \mathbb{D}(E) \setminus$ the control parameter of r;
        insert E' into $\mathcal{E}$;
    Dispatch $\mathcal{E}$ to the executive and ask to choose a new model state E;
return $R(E)$;

There can be multiple flexible control parameters defined in the same action, but the controlled plan management applies with no difference (i.e. still assigned one by one). The rationale behind this assumption is that the domains of these parameters can still be inter-dependent to each other. An iterative assignment ensures that the executive is fully informed about the consequences of each commitment undertaken, so that an exhaustive and uninformed assignment operation can be avoided. In Section 5.3.2, we survey applying the controlled plan management in robot navigation scenario, in which the executive is the motion planner.

Lastly, we formalise the new plan achieved after this process is concluded. We modified the controlled plan definition (see Definition 10) so that it reports specified values for each control parameter of each action in π. We call the updated plan as *a controlled plan with executive decisions (π_{exec})* and it is shown as follows:

$$\pi_{exec} = \{\langle a_i, ts_i, \Delta t_i, \mathbf{r^{a_i}} \rangle \mid \forall i = \{0, \ldots, n\}\},$$

where $\mathbf{r^{a_i}} \in \mathbb{R}$ is a vector denoting the specified values for control parameters of action a_i.

3.5.6 Modifications to The Temporal RPG Heuristic

The Metric Relaxed Planning Graph (RPG) (Hoffmann, 2003) heuristic has been widely used in numeric planning over the last decade. The POPF planner uses an extended variant, based on the Temporal RPG (TRPG), to guide the planner in the search space towards the

Table 3.4: An LP relaxation over a control parameter that does not have a constant upper bound value.

Maximise: $?cash_{(0,0)}$
Subject to:
$bal_0 = 50$
$?cash_{(0,0)} \geq 5$
$?cash_{(0,0)} - bal_0 \geq -\infty$
$?cash_{(0,0)} - bal_0 \leq 0$
$?cash_{(0,0)} + bal'_0 - bal_0 = 0$
$inp_0 = 2$
$inp'_0 - inp_0 - ?cash_{(0,0)} = 0$

goal. The difference between the two heuristics is that the TRPG associates timestamps with each action and fact layer, using rules on relaxed temporal progression to increase the time as the reachability analysis is developed (Coles et al., 2009b).

Our modification to the TRPG heuristic of POPF is to make an additional optimistic assumption: if an action a has a control parameter effect on a variable v, then the control parameter is relaxed to whichever bound in D for the corresponding control variable gives the largest effect (increasing the upper bound or decreasing the lower bound on the reachable values for v). In case the bounds on the control variable depend on the value of state variables (for example, `(<= ?cash (balance ?m))`), then the heuristic constructs an LP at the fact layer that the condition appears, using only the time-independent numeric constraints of the action, to compute the bounds for the heuristic before extracting a relaxed plan. Table 3.4 shows the LP constructed to optimistically approximate the upper bound of the `?cash` variable.

Note that this extension to TRPG can fail to estimate the relaxed bounds of control parameters correctly at every layer of the relaxed planning graph, especially in problems requiring repetitive numeric transfer between variables with control parameters. In Chapter 4, we survey a way of increasing the basic informedness of this heuristic in a wide range of problems with complex numeric transfer.

3.6 Evaluation

The cashpoint domain demonstrates that there are examples of actions in which it is natural to model an action with an infinite domain parameter, other than duration. Furthermore, the `descend` action shown in Figure 3.9 illustrates that there are also natural reasons to transcend the use of a single control parameter, making it impossible to express such models by some compilation of control parameters into flexible durations. This demonstrates that the extended expressive power of the control parameters we propose offers a way to capture important and intuitively significant behaviours that cannot be modelled in existing PDDL formulations.

Although it is clear that the extended expressive power of control parameters has a valuable role to play in modelling realistic domains, there remains the question of whether it is possible to plan with this extended expressiveness. In this section, we consider several

different domains in order to explore the scalability of the performance of POPF when confronting a variety of challenges in the use of control parameters.

Our planner, POPCORN, is a temporal planner that handles discrete numeric change in state variables that is controlled with continuous-valued variables selected by the planner. There are no other PDDL2.1 planners available with similar expressive power to compare on problems using control parameters, including Scotty and cqScotty. Both planners support control parameters for setting *rates of change* for continuous change, but not control parameters used for discrete change, as in the cashpoint example. Therefore, we compare the performance of POPCORN with POPF (the code base on which POPCORN is built). This comparison is carried out by discretising the control parameter choices available to POPF. The purpose of the comparison is to explore the scaling behaviour of POPCORN, since POPF is solving a related and similar problem to the control parameters choice problem, but the discretisation imposes both additional constraints (the possible values of the control parameters are limited) and a simplification (the branching factor is made finite). A comparison with UPMurphi (Penna et al., 2009) or its close relative, DiNo (Piotrowski et al., 2016), is also possible, using discretised control parameters (which could be managed automatically in UPMurphi within its existing machinery). However, its performance does not scale well in domains with long combinatorial chains, such as the Procurement and Terraria domains.

In addition to discrete comparison with POPF, we make an approximate comparison with Scotty on domains with control parameters that are rates of change (using problems from (Fernández-González et al., 2015a)). Since POPCORN does not handle non-linear interaction between the rate and the evolving time, we modified these domains so that the rates (i.e. the control parameters) and the duration parameter form linear constraints placed in duration constraints field (and the resulting models are run on POPCORN). This modelling technique is discussed in Section 3.5.3 and an example encoding is given in Figure 3.9. Although the time is not continuously evolving in the POPCORN domains, the duration is flexible and it is in linear interaction with the rates. All domain models and problem instances used in our experiments can be found online: https://github.com/Emresav/ECAI16Domains

Further in this section, we individually describe the domain models used in our experiments, identify their characteristics from the planning as search perspective and their complexity as temporal and numeric problem. We also describe how the problem instances of each domain is generated and varying complexity levels of the problem instances are achieved. Then, we present the experimental results and analyse them with respect to various planning perspectives including the solution time, the makespan, the mean scheduling time per state and the number of states evaluated by each planner.

3.6.1 The Cashpoint Domain

This domain is based on the motivating example presented in Section 3.2. The cashpoint domain creates a relatively flat search space, with short plans.

The purpose of using this domain is to confirm that POPCORN effectively generates plans with control parameters. As the amount of cash required in the goal is increased

relative to cashpoint limits, the plan has to include additional withdrawal actions. This is expected in the plans generated by POPF as the discretised control parameter choices can result in multiple instances of the withdrawal action in the plan. Also, it is important that the planners realise that the plan cannot achieve the goals with a single withdrawal (or, more generally, insufficient withdrawals) early enough to avoid wasted search effort.

We extend the basic actions of the original domain to include multiple currencies, allowing new currencies to be obtained using an exchange bureau. This complicates problems by constructing chains of constraints connecting the values of control parameters across multiple actions.

In order to make a comparison with POPF, we add a set of `WithdrawCash` actions (to the POPF domain) with fixed withdrawal values, namely `?cash` $\in \{1, 5, 10, 20\}$. This assumption allows us to make a fair comparison in these domains with POPF, which does not aim to handle control parameters. Regardless of the values we fix for the withdrawal values, the POPF will usually generate longer plans (i.e. when the problem requires `(inpocket ?p)` $\leq$ `£20`). POPCORN does not require any fixed value, so it is able to solve the problem for any value of `(inpocket ?p)` within the bounds defined.

In all domain models we include in these experiments (including this one), we use a set of 12 problem instances. The numeric goal limits and the types of currencies are increased, and in some problem instances, the planner is forced to exchange currencies at the exchange bureau.

3.6.2 The Procurement and Terraria domains

The procurement domain is about finding a plan to manufacture final products, which require different amounts and types of raw materials to be procured from a store and intermediate products to be produced at a workshop. The purpose in defining this domain is to explore the effects of scaling numbers of control parameters in individual actions. Materials can be purchased in a single action, selecting the numbers of each of the materials available at the supplier to be included in the single purchase. This problem also contains a deeper search space, producing longer plans, allowing us to explore the cost of solving increasing numbers of more complex constraint sets.

The complexity of each instance is increased by extending the depth of the bill of materials tree. Figure 3.14 shows the most challenging problem instance in which two of product A, a product B, and a product L are to be delivered to customers Customer1, Customer2, and Customer3, respectively. The leaf nodes represent raw materials, which can be obtained from the store. The other nodes represent the intermediate and final products. To produce a product A, we need to have already sub-assembled two product B and a product C. These intermediate products require products I, J, K, D, and E. In this domain, the batch sizes of items procured and produced are taken as control parameters, whereas these parameters are discretised with the values of 1, 5, 10, 20 for the test runs in POPF.

The Terraria domain is similar to the procurement domain, but it additionally includes a capacity constraint that limits the total amount of raw material procured. As an extension of the procurement problem set, the complexity of each Terraria problem instance is extended by decreasing the capacity limit of raw materials. The purpose of this test is to introduce a

constraint that links multiple control parameters within a single action.

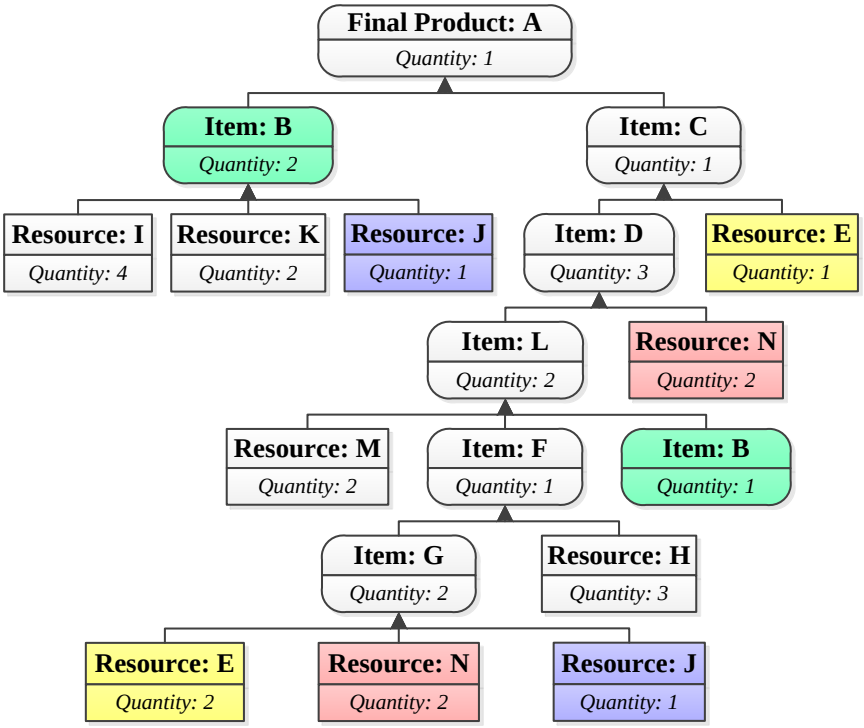


Figure 3.14: Bill of material tree of product A in procurement domain model. Coloured nodes indicate that those items are already used in the tree.

3.6.3 The 2D-AUV-Power domain

This domain is based on the Kongming AUV domain presented in Section 3.5. The purpose in introducing this domain is to show that duration and other control parameters in an action can be successfully constrained within a single constraint.

In this domain, the AUV travels between different waypoints to collect samples in a two dimensional underwater environment (depth and horizontal distance). Figure 3.9 shows the `descend` action with our proposed syntax for control parameters. In this experiment, we reinterpret this action as a single `glide` and allow the velocity range to include positive and negative changes in the y-direction (depth), so that the action can be used to ascend or descend. The duration of `glide` is constrained by the maximum and minimum displacement rates of the vehicle in x- and y-directions. The linear power constraint of `(<= (+ (*?dx 3) (*?dy 4)) 90)` limits the total power use of the vehicle across the two motors in gliding. The complexity of this experiment increases by increasing the sample destinations and decreasing the power capacity (per glide action) of the vehicle. The displacement rate of the glide action is fixed with 1, 5, 10 and 20 units (`?dx`, `?dy` $\in \{1, 5, 10, 20\}$). The AUV can take a sample whenever the x- and y-positions are in the range the sample destination.

3.6.4 Experimental Results

Figure 3.16 shows the time taken to solve each problem instance, and the number of states evaluated in the domain models. Observe that POPF generates more states than POPCORN does for almost all problem instances. The reason behind this behaviour is the existence of wide plateaus in the search space of POPF, as we defined multiple action schemas (each defined with fixed control parameter values) for each action that includes at least one control parameter. For instance, in the cashpoint example, we defined 4 different `WithdrawCash` actions in the POPF domain, whereas we have only one of it in the POPCORN domain. Consequently, the POPF planner generates 4 times more `WithdrawCash` grounded actions for a problem instance than the POPCORN does at any search depth where the action is applicable. This fact consequently affects the solvability of the POPF planner as it can result in an exhaustive search.

As discussed in Section 3.5.5, the LP avoids early commitment to values of control parameters, but maintains the constraint space in which they must lie. However, a deeper search space might lead to an excessive use of the LP during planning. As mentioned earlier in this section, the procurement domain model creates a deeper search space than the cashpoint domain. We observe that POPCORN takes longer than POPF to generate plans in almost all of the procurement problem instances. The reason behind this behaviour is that POPF realises the use of its Simple Temporal Network (STN) is sufficient to check state consistency in most of the steps, whereas POPCORN mostly relies on solving LP models for this purpose. Coles et al. show that using the STN in temporal-numeric domains is significantly faster than using the LP (Coles et al., 2012).

Figure 3.15 shows the mean time spent solving the LP at each state. These figures indicate that the constraint-solving time per state explored by POPCORN gradually increases, because larger-sized linear programs are generated as the problem complexity increases. Observe that the average mean time spent on the procurement domain is generally higher than the times on the cashpoint domain evaluated by both planners. The rationale behind this fact is that the procurement domain induces a chain of numeric subgoals to satisfy the top-level numeric goal of each problem. This feature of the procurement domain has two immediate consequences in the planning process. First, the planners spend more time constructing the chain relationship in the heuristic. Second, these subgoals create larger-sized linear programming constraint sets, so that the planners (indeed, it is only POPCORN) spend more time computing the LP model.

Despite the LP costs it is worth noting that POPF generates poor plans. Repetition of the same action is observed in plans generated by POPF. Even though multiple choices of the same discrete actions are defined, POPF usually chooses the ones with the least increase or decrease effects, which leads it to generate poor plans; POPCORN generates better plans where repetition of the same action is not usually observed. Figure 3.17 compares the solution quality for each problem instance solved by POPCORN and POPF.

The Scotty planner introduces various domain models with flexible rates of change (that are control parameters) in continuous effects. Table 3.5 compares the performance of POPCORN and Scotty in these domain models. As previously said, we avoid non-linear interaction between control parameters by modelling the problems in the same fashion

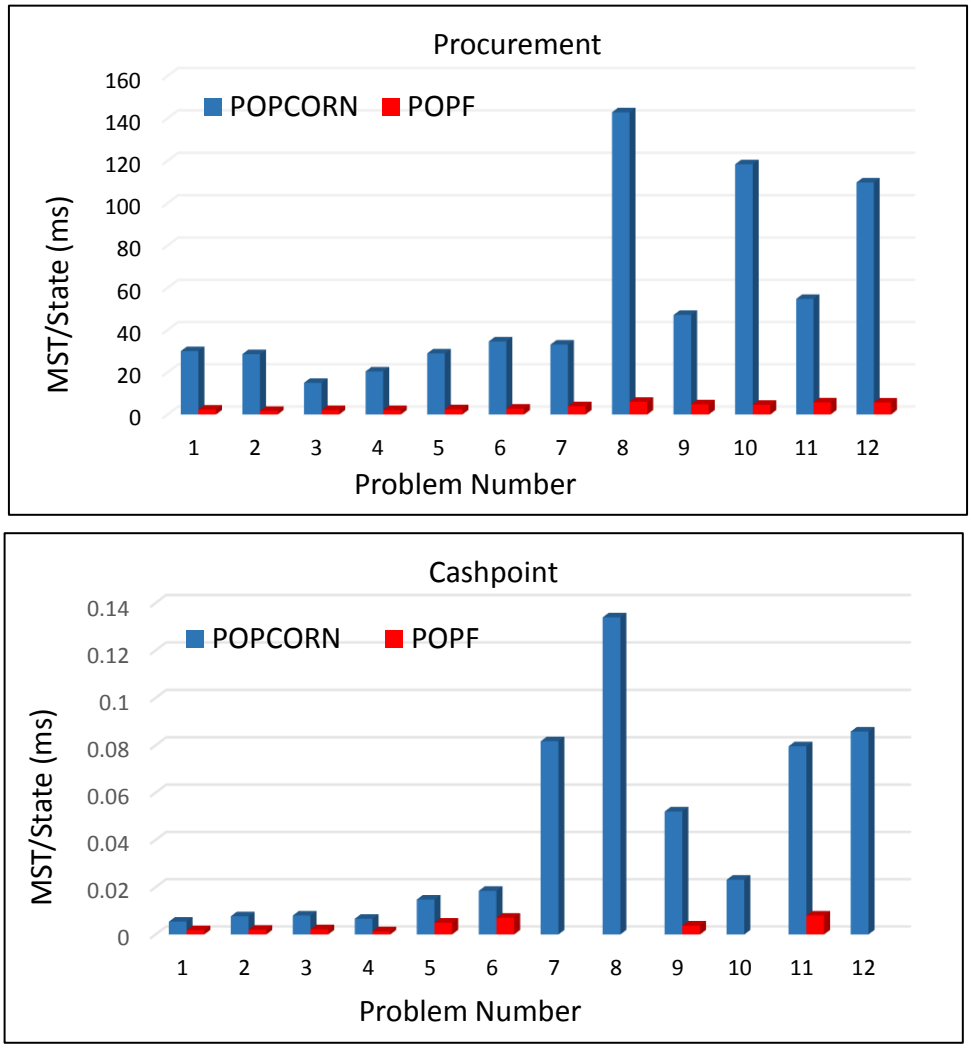


Figure 3.15: Mean Scheduling Time (MST) per state in procurement and cashpoint instances.

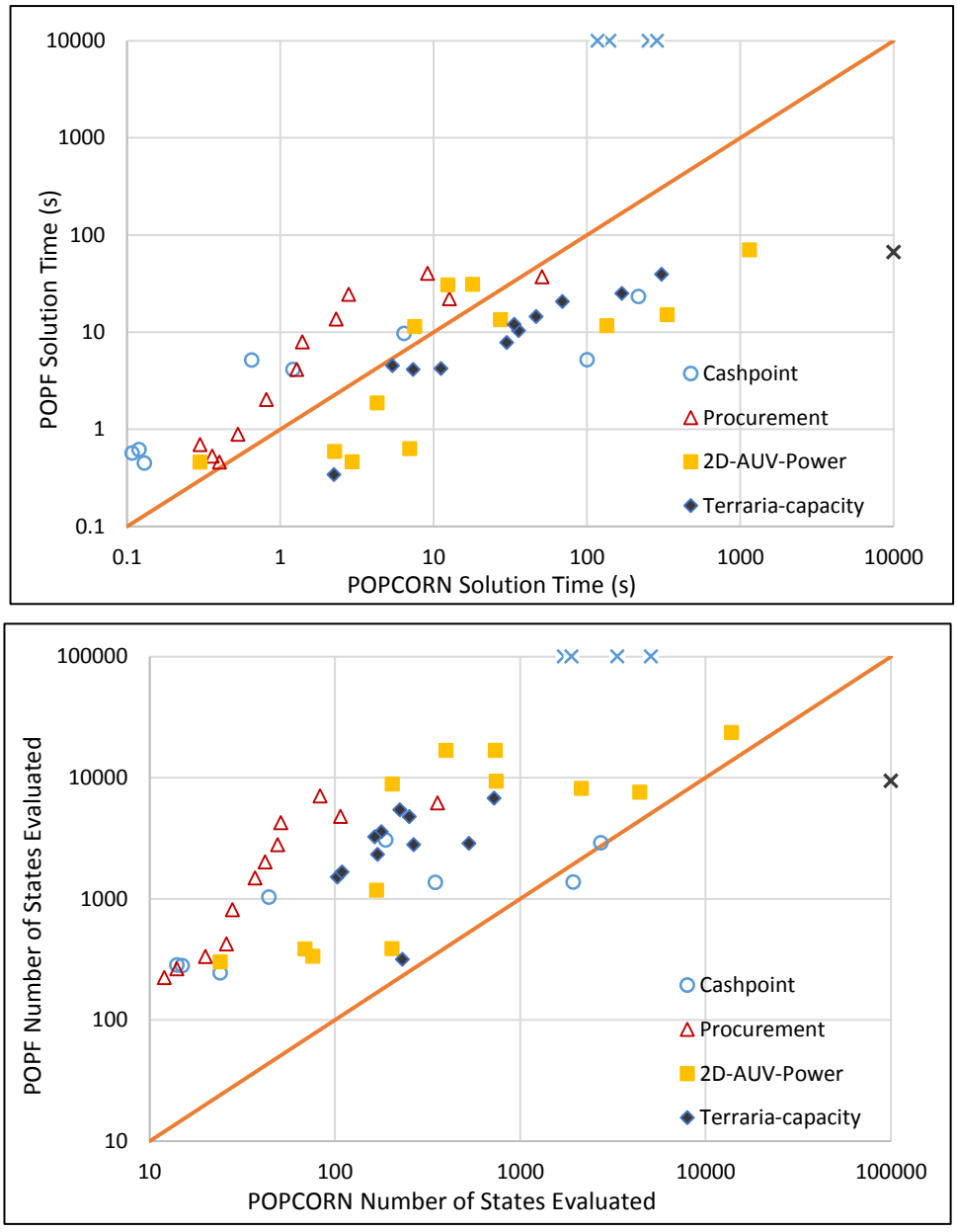


Figure 3.16: Comparison of time taken by POPF and POPCORN to solve each problem instance, and the numbers of states evaluated, in four domains. The crossed points indicate that only one of the planners found a solution.

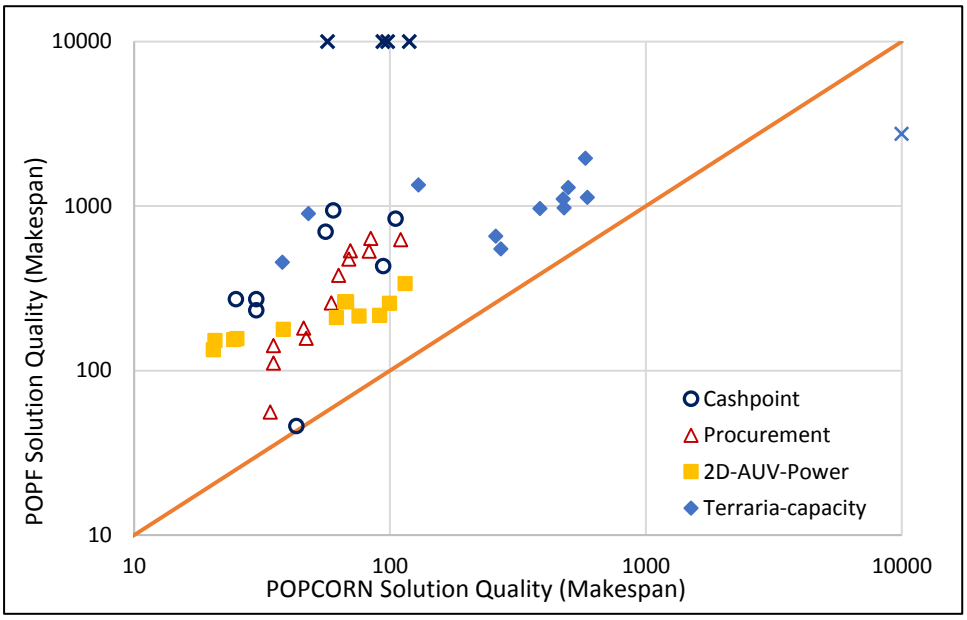


Figure 3.17: Comparison of plan quality (measured in this case as plan makespan) in four temporal numeric domains. POPCORN is compared with POPF on each problem instance.

Table 3.5: Comparison between POPCORN and Scotty in the 5 problems in Firefighting, 2D and 3D AUV navigation (Fernández-González et al., 2015a). The numbers in the brackets are the makespan, while the numbers outside the brackets are the execution times.

Domains:	2D AUV1	2D AUV2	3D AUV	Firefighting1	Firefighting2
Scotty	2.6(139)	8.4(260)	0.8(12)	1.2(20)	2.5(14)
POPCORN	0.9(116)	8.8(153)	0.5(12)	0.1(20)	0.7(15)

discussed in Section 3.5. 2D- and 3D-AUV domains are based on an underwater sampling mission scenario. In the 2D-AUV domain, the vehicle collects sampling data at the same depth (i.e. water depth), whereas in the 3D-AUV domain the vehicle collects data in different depths. In this experiment, we slightly modified these domains to challenge the performances of both planners. Seven and fourteen sampling destinations are defined in 2D AUV1 and in 2D AUV2 domains, respectively. We observed that Scotty generates longer plans in 2D AUV domains than POPCORN due to taking longer paths in both problems. The difference on the plan quality presumably results from the fact that the Scotty planner makes an early commitment on the control parameter values when finding a fixed plan, and this consequently leads to repetitive execution of the same action. We suspected to observe this sacrifice on the plan quality in Section 3.3. The difference in the execution times is insignificant as both systems rely on solving linear programs during similar forward state space search planning.

In this chapter, we surveyed a detailed treatment of control parameters in terms of how they are conceptually represented in temporal-numeric planning problem, how they are encoded in a standardised modelling language and handled in forwards heuristic search. We proposed an optimistic (yet somewhat unrealistic as it ignores temporal dimension of the problem) extension to the TRPG heuristic, and in these experiments, we observed that

this heuristic extension is deemed insufficient in numerically challenging problems. In the procurement domain, for instance, the numeric resources are repeatedly consumed and/or produced with control parameters, and a top-level numeric goal can necessitate satisfying a set of numeric subgoals. This problem is rather difficult for a heuristic with various useful relaxations (e.g. delete-relaxation, infinity analysis). The rationale behind the insufficiency of the heuristic is that considering each action to apply arbitrarily many times (i.e. the infinity analysis) during graph expansion makes the variables unbounded, even if the use of LP provides somewhat strong relaxed bounds.

As previously said, the complexity of the problem instances is directly related to the complexity of the numeric goals, and the ones we used in these experiments were fairly simple to achieve. As the numeric goal complexity is increased (in problems of most domain models used in this experiments), the heuristic can promote false activities as helpful actions (previously described in Section 2.5.2.2). The rationale behind this behaviour is that the heuristic fails to provide strong numeric bounds on variables. Consequently, the heuristic can lead the search into a dead-end in certain conditions. We identified that the heuristic we use, the extended TRPG, needs structural revision on its numeric satisfaction mechanism that aims to provide a certain level of informedness in complex numeric problems. We describe the core of the numeric resource transfer issue and propose a new extension to the TRPG heuristic in the next chapter.

3.7 Summary

Physical and logical properties of real world examples require multiple numeric variables to create realistic planning models. We have shown examples of problems, in which it is most natural to model a choice of numeric parameter values to control the behaviour of actions, in a similar way, but in addition to, the duration of flexible-duration actions. Furthermore, the opportunity to combine multiple control parameters in a single action, so that the control space is multi-dimensional, is motivated by specific examples.

We have presented a planning approach capable of solving problems in domains with control parameters and we have demonstrated that it is capable of solving interesting problems with a range of characteristics, performing scalably and producing efficient plans. We compare performance with two alternatives: POPF using discretised control parameters and Scotty, which offers a different role for control parameters that does not include examples such as the cashpoint, procurement or Terraria domains.

The approach we have proposed, in which constraints governing control parameters are accumulated into a constraint program that is checked for feasibility as states are progressed, generalises to other types and to non-linear constraints, subject to the capabilities of the solver for the constraint program. We intended to explore this in future work.

We have presented a highly optimistic way of handling control parameters in the heuristic of our base system, the TRPG. We identified that the existence of control parameters creates rather complicated issues in the heuristic evaluation that needs to be identified and handled delicately. In the next chapter, we recondition the heuristic used in this chapter to solve problems that are deemed highly challenging or unsolvable by our system.

Chapter 4

Refined Infinity Analysis for Planning with Control Parameters

4.1 Introduction

Although control parameters are a modelling requirement in real world planning applications, planning with control parameters is rather challenging as a search problem. Their existence changes the structure of the search space by creating an infinite branching choice of applicable actions at a state. We described the resulting state representation, the $C-$state, in the previous chapter. When at a $C-$state, the extended TRPG heuristic (presented in Section 3.5.6) can provide an uninformative heuristic estimates as the use of infinity analysis immediately makes the relaxed bounds of variables unbounded, even if the minimal use of LP in the heuristic provides somewhat strong relaxed bounds. Consequently the heuristic fails to prune unreachable actions (in EHC) and in most cases it leads the search to a dead-end. This issue is observed in producer-consumer planning problems, in which numeric resources are repeatedly consumed (and unavailable for other uses) and produced with control parameters (e.g. the procurement and the terraria domains). Delete-relaxation based heuristics (except the LP-RPG) struggle solving these sort of problems (with or without control parameters), as they ignore the consumption of numeric resources.

To aid in illustration of this point, consider a simplified linear Generator domain (Howey and Long, 2003), in which the limited resource (i.e. the fuel level) is repeatedly transferred (produced by the *refuel* action and consumed by the *generate* action) between states using actions with control parameters. The generator runs on fuel and in return, it produces energy. The goal is to have a certain level of energy produced while the generator has to have fuel to operate. The heuristic fails to include the refuel action in the helpful actions set as it ignores consumption of the fuel level, and instead it promotes the generate action as helpful (even if it must include the refuel action). This has been previously identified as a common issue in producer-consumer problems by Coles et al. (Coles et al., 2008b), they called *helpful action distortion*. Another common issue of the delete-relaxation based

heuristics (including the heuristic of the Scotty and the cqScotty planners) is *cyclical resource transfer*, which is defined as follows:

> "The phenomenon of Cyclical Resource Transfer (CRT) is a consequence of the encoding of actions that move resources around, combined with the relaxation of negative effects. To encode movement of a resource, it is removed from one location and added to another. The removal is encoded as a decrease and this is relaxed when building the Metric RPG. As a result, moving resources appears to generate new resource at the destination, making movement a spuriously attractive alternative to production. " (Coles et al., 2013, pg. 357).

One way to avoid these weaknesses is to eliminate the infinity analysis (i.e. there can be arbitrarily many layers in the graph) from the heuristic and use an optimisation tool to compute the true bounds of every state variable (affected by a control parameter) at every fact layer of the RPG for every state reached (generalising, in some way, the behaviour of LP-RPG (Coles et al., 2013)). This is costly, however. The number of LP calls per RPG graph would be equal to the multiplication of the number of control parameters, numeric variables, actions and layers. Analysis of LP-RPG indicates that the cost of multiple LP solving in the heuristic evaluation of states dominates the computation cost for the heuristic (Coles et al., 2013). The trade-off between obtaining strong max/min bounds on variables, and the cost of computing them, is critical. Each call to the optimisation tool (in our case it is LP-solve) carries a cost and excessive use of it can make the whole approach inefficient.

In this chapter, we extend a variant of the TRPG heuristic (Coles et al., 2009b) (presented in Section 3.5.6) that guides the state progression of C-states alongside the P-states. Our approach combines the discretisation of alternating layers approach used in the RPG heuristic of the Metric-FF (Hoffmann, 2003) with infinity analysis used in the TRPG to form a middle ground between the two. We call this approach *refined infinity analysis*. We identify that such a hybrid planning graph approach improves the basic informedness of the heuristic when planning with control parameters (especially when there is producer-consumer relationship between actions). Compared to other approaches we avoid making use of linear programming (LP) as it provides an expensive solution for a heuristic computation. We investigate the numeric interval relaxation of the state variables that are affected by at least one control parameter.

The structure of this chapter is as follows: we describe a motivating problem based on the procurement example before providing technical detail about the contribution in Section 4.2. We examine the producer-consumer problem in depth in Section 4.3 and in Section 4.4. Then we introduce our proposed extension to the TRPG heuristic in Section 4.5. We show that our extension is fast and scalable in a domain-independent way in Section 4.6. Finally, we conclude with discussion and conclusion in Section 4.7.

4.2 An Example Problem

We mentioned that the procurement domain is a rich producer-consumer problem. We revisit this domain to use it as our running example here as follows. Suppose that we add various items together (different types and quantities) to produce intermediate and finished

products in a workshop. The final and intermediate products are assembled at the workshop, while the raw materials are outsourced. Figure 4.1 shows the material requirement tree used in the manufacturing process of the item a. In the figure, item a is the final product, items c, b, and d are intermediate products (white circular nodes), items g, f, e, i, h are the raw materials (gray square nodes). In order to assemble one item b, we must have at least two d, one f and three e in stock. In case these items are not available, the planner should consider acquiring them: producing d, and supplying e and f.

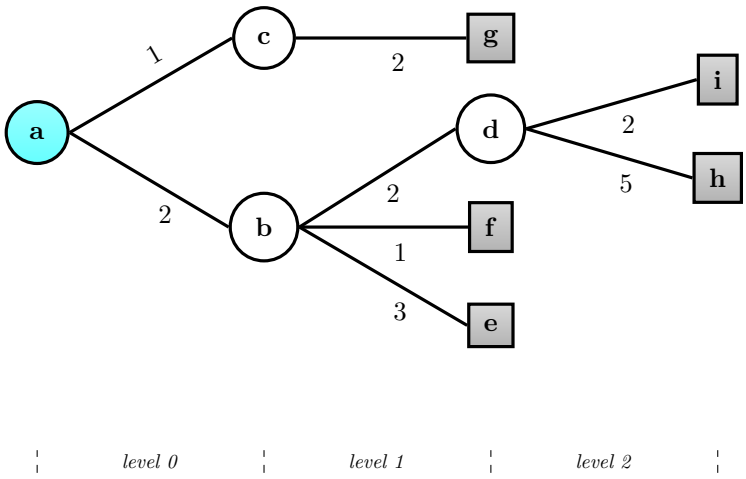


Figure 4.1: The material requirement tree for the motivating example.

This example is a simplified version of the procurement domain we introduced in Section 3.6. We eliminated the location objects from the original domain so that the example is only concerned with producer-consumer behaviour between actions. In addition, the duration of each action is fixed for simplicity. We use a far more complex version of this domain in our experiments (see Section 4.6.1.2). The main actions of this example are given in Figure 4.2. In this domain each action has a single control parameter, `?unit`, which controls the amount of item assembled or supplied per execution of that action. The `?unit` parameter takes its value from an infinite domain of $[0, 10]$, however it is also constrained by the stock level of other items. For instance `(>= (stock ?b) (* ?unit 2)))` condition limits the value of the control parameter can take in *assemble_A* action. For this reason, the upper bound of `?unit` can change at every state, which makes it a non-constant bounded control parameter (a formal definition of it will be given in the next section).

In this problem the goal is to achieve a stock level of 20 for the final product a: `(a` $\geq$ `20)`. The initial stock level of the resources are: 2 of item a, 3 of b, 3 of c, 2 of d, 4 of e, 1 of f, and the stock level of the rest is zero. Since the numeric goal on a is greater than its stock level, we need to assemble more of item a. However, the execution of *assemble_A* action depends on the stock levels of item b and c, while they depend on stock levels of items d, e, f, and g, and so on. Thus, this process requires cascading arrangement of producer-consumer actions (i.e. assemble and supply actions).

Figure 4.3 shows the modified TRPG reachability graph of POPCORN presented in Section 3.5.6 and the solution extraction of the problem. The behaviour of the heuristic in this example is important. For each non-constant bounded control parameter appearing

```
(:durative-action assemble_A
:parameters (?a - itemA ?b - itemB ?c - itemC)
:control (?unit - number)
:duration (= ?duration 2)
:condition (and (at start (>= ?unit 0)) (at start (<= ?unit 10))
 (at start (>= (stock ?b) (* ?unit 2))) (at start (>= (stock ?c) ?unit))
 (at end (>= (stock ?b) 0 )) (at end (>= (stock ?c) 0 )))
:effect (and (at end (increase (stock ?a) ?unit))
 (at start (decrease (stock ?c) ?unit))
 (at start (decrease (stock ?b) (* ?unit 2)))))

(:durative-action assemble_B
:parameters (?b-itemB ?e-itemE ?f-itemF ?d-itemD)
:control (?unit - number)
:duration (= ?duration 1)
:condition (and (at start (>= ?unit 0)) (at start (<= ?unit 10))
 (at start (>= (stock ?e) (* ?unit 3)))
 (at start (>= (stock ?f) (* ?unit 1)))
 (at start (>= (stock ?d) (* ?unit 2)))
 (at end (>= (stock ?f) 0 )) (at end (>= (stock ?e) 0 ))
 (at end (>= (stock ?d) 0 )))
:effect (and (at end (increase (stock ?b) ?unit))
 (at start (decrease (stock ?e) (* ?unit 3)))
 (at start (decrease (stock ?f) (* ?unit 1)))
 (at start (decrease (stock ?d) (* ?unit 2)))))

(:durative-action assemble_C
:parameters (?c - itemC ?g - itemG)
:control (?unit - number)
:duration (= ?duration 1)
:condition (and (at start (>= ?unit 0)) (at start (<= ?unit 10))
 (at start (>= (stock ?g) (* ?unit 2))) (at end (>= (stock ?g) 0 )))
:effect (and (at end (increase (stock ?c) ?unit))
 (at start (decrease (stock ?g) (* ?unit 2)))))

(:durative-action assemble_D
:parameters (?d - itemC ?h - itemH ?i - itemI)
:control (?unit - number)
:duration (= ?duration 1)
:condition (and (at start (>= ?unit 0)) (at start (<= ?unit 10))
 (at start (>= (stock ?h) (* ?unit 5)))
 (at start (>= (stock ?i) (* ?unit 2)))
 (at end (>= (stock ?h) 0 )) (at end (>= (stock ?i) 0 )))
:effect (and (at end (increase (stock ?d) ?unit))
 (at start (decrease (stock ?h) (* ?unit 5)))
 (at start (decrease (stock ?i) (* ?unit 2)))))

(:durative-action supply_raw_material
:parameters (?d - item)
:control (?unit - number)
:duration (= ?duration 1)
:condition (and (at start (>= ?unit 0)) (at start (<= ?unit 10))
 (at start (>= (stock ?d) 0)) (at start (can_supply ?d)))
:effect (and (at end (increase (stock ?d) ?unit))))
```

Figure 4.2: Main actions of the example domain encoded in PDDL extension we introduced in Chapter 3.

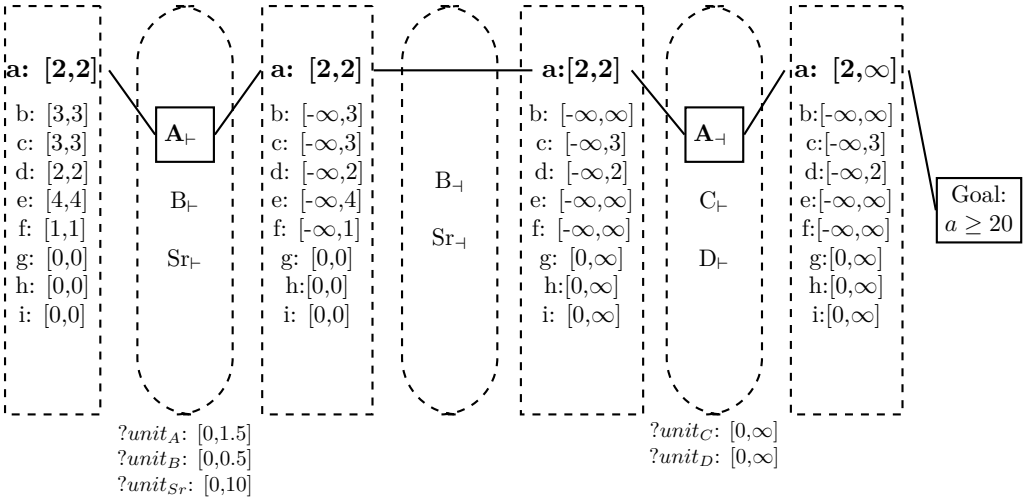


Figure 4.3: The extended TRPG reachability graph of POPCORN. We represent *assemble_A*, *assemble_B*, *assemble_C*, *assemble_D*, and *supply_raw_material* actions as A, B, C, D and Sr, respectively. The relaxed upper bounds of $?unit_A$, $?unit_B$, $?unit_C$, $?unit_D$ are computed by the LP, whereas $?unit_{Sr}$ is not.

in the graph, it constructs an LP model (ignoring temporal variables and constraints) and solves it to find their relaxed upper bounds. Table 4.1 shows the LP model of the `?unit` parameter of assemble_A, and the optimal solution of this model is $?unit_{(0,0)} = 1.5$. The heuristic sets this as the upper bound of the parameter at the first layer the action appears and never updates it again in further layers.

The heuristic infers that all actions can be applied infinitely many times once they become applicable during the graph expansion (i.e. the infinity analysis). Although the LP provides somewhat strong relaxed bounds on these parameters, applying each action infinitely many times makes the relaxed bounds of the affected variables unbounded (e.g. the relaxed upper bound of a immediately becomes ∞). For this reason, the use of LP becomes almost useless as its decision is suppressed by the infinity analysis.

When satisfying the numeric goal, $a \geq 20$, in solution extraction phase, the goal is assumed to be reached without considering actions that replenish its resources (that are *assemble_B*, *assemble_C*, *assemble_D*, and *supply_raw_material* actions). The heuristic should have inferred that there is a set of sub-goals that must be satisfied in parallel (that are indeed analogous to the numeric landmarks of (Scala et al., 2017)), and these sub-goals are (based on resource requirements of the goal): $b \geq 40$, $c \geq 20$, $d \geq 80$, $e \geq 120$, $f \geq 40$, $g \geq 40$, $h \geq 400$, and $i \geq 160$. This behaviour of the heuristic (disregarding the sub-goals) can eventually lead the search to a dead-end. As mentioned previously, this is a familiar problem for delete-relaxation based heuristics and it is called helpful action distortion. The problem comes from the fact that there is insufficient communication, in the graph expansion, between the variables in the numeric preconditions of actions and the constraints on the variables in their effects. Considering the numeric constraints of the problem, we expect that the assemble_A action can only occur once, and then it can only occur again when its numeric resources are fully replenished. The helpful action distortion problem is not only restricted to numeric domains. It can be observed in more general

Table 4.1: The LP relaxation model called by the extended TRPG at the fact layer zero to find the relaxed upper bound of `?unit` of assemble_A action. The construction of this model is described in depth in Section 3.5.6.

Maximise: $?unit_{(0,0)}$
Subject to:
$0 \leq ?unit_{(0,0)} \leq 10$
$0 \leq b_0 - 2{\cdot}?unit_{(0,0)} \leq +\infty$
$0 \leq c_0 - ?unit_{(0,0)} \leq +\infty$
$a'_0 - a_0 - ?unit_{(0,0)} = 0$
$b'_0 - b_0 + 2{\cdot}?unit_{(0,0)} = 0$
$c'_0 - c_0 + ?unit_{(0,0)} = 0$
$a_0 = 2$
$b_0 = 3$
$c_0 = 3$
$a'_0, b'_0, c'_0 \geq 0$

planning as search problems. Xie et al. describes the problem of choosing wrong activity as helpful early in the forwards search as *early mistakes* (Xie et al., 2014; Xie, 2016), which can not be recovered from as it can require significant backtracking search.

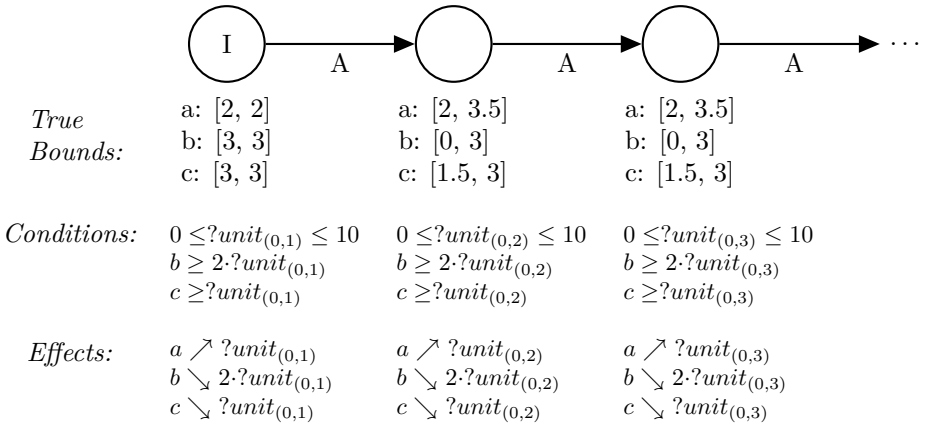


Figure 4.4: The dead-end in forward search due to poor heuristic guidance. I: initial state.

Figure 4.4 illustrates how the forward search is led to a dead-end when the heuristic repetitively declares the assemble_A as the only helpful action during search. The true bounds are obtained from the LP solver when checking the state consistency, and they are subject to numeric constraints that appears in conditions and effects. The variables in these constraints are encoded in the form described in Section 3.5.4. The start-end snap-actions are compressed in this figure. Observe that whenever the lower bound of b hits zero (where it is initially 3), the true bound of a stops increasing (based on the solution obtained from the LP with accumulated constraints). We consider b as the *bottleneck resource* of assemble_A at that state. This detail plays a key role in answering the following questions about repetitive execution of assemble_A without requiring replenishment:

– How many times can this action be applied without reaching to a dead-end?

– What is the total numeric contribution of this action on satisfying the numeric goal over variable a at a state s?

We identified that tackling these questions in the heuristic is critical to construct a hybrid planning graph (i.e. a combination of discretised and continuous alternating layers) that we propose to achieve in this work. Addressing these questions would allow us to attain some sort of numeric inference between resources. We aim to improve the numeric informedness of the heuristic by monitoring how many times such actions can be executed. To achieve this, we indirectly keep the bottleneck resources under surveillance until the corresponding numeric goal is satisfied. Another challenge this problem poses is that the planner should find a plan that minimises the required amount of resources. Thus, a realistic estimate on how many times such actions can be executed is critical to obtain a good quality solution plan. We address this issue in detail in Section 4.5.1.

4.3 Preliminaries on Producer-Consumer Behaviour

We briefly exemplified the producer-consumer behaviour arising in a simplified linear Generator domain in Section 4.1 and in the motivating example in Section 4.2. We can generalise the behaviour as follows. The producer-consumer behaviour is observed at a planning problem where the goals of the problem impose the numeric resources to be repeatedly decreased and increased (produced and consumed) when progressing through states. We can observe the behaviour in numerous real world applications: manufacturing of goods, trading commodities, energy conversion, chemical reactions, any application that requires resource transformation, and so on.

In recent years the constrained producer-consumer behaviour in task planning has been thoroughly discussed. For instance, Coles et al. (Coles et al., 2013) explored the behaviour in which resource transfer occurs using variables that take their values from finite domains, whereas Laborie explored an analogous behaviour in scheduling (Laborie, 2003). Scala et al. extended the h^{max} heuristic to capture implicit subgoals in general numeric planning (Scala et al., 2016b). The work of Coles et al. reflects only a restricted version of real world producer-consumer problems, where the variables only have a few numeric choices. For instance, the fuel amount is restricted to be taken from a domain of $\{5, 10, 50, 200\}$ in a refuel action. In case the required fuel amount is 75 units, the planner can achieve this at once by refuelling it 200 units or it can choose to apply a few instances of the refuel actions by smaller amounts (i.e. 5 or 10 units). Both options produce a poor quality solution as it is desirable to minimise the use of resources and the makespan. In this work we explore the behaviour when the variables are not restricted to take their values from a few points, but from a continuous surface of points. This is achieved with the help of control parameters in resource transfer actions.

In this section we first define various *producer* and *consumer* actions that we consider in this work. We split consumer actions into two categories: *constant* and *non-constant* bounded consumer actions. This categorisation is an intuitive generalisation to such actions that differ based on how the control parameters are used in the resource constraints. Using these definitions we then identify producer-consumer variables.

It is worth mentioning that the definitions we present in this section are derived from the pre- and post-conditions and effects of actions in the domain model. Thus, the classification occurs at the parsing stage (of the domain and problem file) in our implementation (which is prior to initialisation of search and heuristic algorithms). We mention the occurrence limits of actions (k) and the earliest layer (which is time-stamped) that the action appears, l, in the definitions. Although the classification of these actions does not change during planning, the k and l entities are the local variables of the heuristic, so that they are updated every time we generate a new relaxed plan at a state s. We examine the computation of k in depth in Section 4.5.

4.3.1 Producer-Consumer Effects and Fluents

Before we identify the producer-consumer behaviour in grounded action level, we survey individual numeric effects and fluents of actions that are important during constrained resource transfer. Breaking producer-consumer actions into its components (as its numeric effects and variables) holds the key role of identifying the problem in the layer-based planning graph as they make the numeric transition between fact and action layers. In this section we define the produced-consumed resources and the producer-consumer effects as we frequently refer to these terms throughout the section.

Let $\mathbf{v_p} \subseteq \mathbf{v}$ and $\mathbf{v_c} \subseteq \mathbf{v}$ are vectors of state variables. $\mathbf{v_p}$ denotes a vector of produced resources and $\mathbf{v_c}$ denotes a vector of consumed resources in a planning problem. We define produced-consumed resources as follows:

Definition 12 (Produced-Consumed Resources) *A numeric variable $v_p \in \mathbf{v_p}$ is a produced resource if and only if any instantiated action a can monotonically increase v_p. A numeric variable $v_c \in \mathbf{v_c}$ is a consumed resource if and only if any instantiated action a can monotonically decrease v_c.*

It is important that, during graph expansion and extraction, consumed resources should be considered to be replenished to avoid common heuristic problems. Thus, in this work we heavily refer to these terms when we identify the switch points between finite and infinite analysis.

We define a slightly modified version of numeric effects that shows producer and consumer behaviour, as follows:

Definition 13 (Producer-Consumer Effects) *Suppose that $\mathbf{v}$ and $\mathbf{d^a}$ are vectors of numeric variables (size of n) and control parameters of an action a (size of m), respectively. A numeric effect of an instantiated action a, $\text{eff}_y(a)$, is a producer (or a consumer) if and only if it monotonically increases (or decreases) a numeric variable $v_i \in \mathbf{v}$ with an arbitrary function of $f(\mathbf{v}, \mathbf{d^a})$, where $y \in \{\vdash, \dashv\}$ and m, n are constant positive integers. Each producer-consumer holds the following entities:*

- `ID` is the unique identifier of $\text{eff}_y(a)$.
- `op` is the assignment operator of $\text{eff}_y(a)$, where $\texttt{op} \in \{\nearrow, \leftarrow, \searrow\}$
- `actID` is the unique identifier of the instantiated action of $\text{eff}_y(a)$.

– `fluentIndex` is the index of the produced (or consumed) variable v_i.

– `vars` is an array of indices of the numeric variables and control parameters that affects v_i that appear in linear function of $f(\mathbf{v}, \mathbf{d^a})$.

– `weights` is an array of constant numeric coefficients of `vars`.

This definition has the following consequences:

– The consumed-produced resource (v_i) of the effect is recorded as the `fluentIndex` of the effect. This index distinguishes these resources from other numeric variables appearing in $f(\mathbf{v}, \mathbf{d^a})$.

– Each numeric effect affected by an action-specific parameter (i.e. a control parameter) becomes action-specific. Thus, such effects hold a unique action identifier, too.

4.3.1.1 Evaluating Relaxed Minimum and Maximum Values of Numeric Effects

Suppose that $\mathbf{v}$ and $\mathbf{d^a}$ are vectors of numeric variables (size of n), the control parameters of an action a (size of m), respectively, where $c \in \mathbb{R}$, $w \in \mathbf{w}$ and $\mathbf{w} \in \mathbb{R}^{m+n}$. All variants of the Metric-FF planning system, including POPCORN and its predecessors, convert the planning task into a *linear numeric task*. The numeric expressions are in a restricted form (defined as Linear Normal Form (LNF) (Hoffmann, 2003)), where each is a weighted sum of variables plus a constant. On the grounds of this restriction, we formulate the numeric expression $f(\mathbf{v}, \mathbf{d^a})$ as a linear function over $\mathbf{v}$ and $\mathbf{d^a}$, as follows:

$$f(\mathbf{v}, \mathbf{d^a}) = w_1 \cdot v_1 + \cdots + w_n \cdot v_n + w_{n+1} \cdot d_1^a + \cdots + w_{m+n} \cdot d_m^a + c$$

A numeric effect affects its fluent v with a linear function that consists of various state variables, control parameters (i.e. $f(\mathbf{v}, \mathbf{d^a})$) and each holds different upper and lower relaxed bound limits in the heuristic. Applying a numeric effect updates the relaxed bounds of its fluent v by the aggregated minimum and maximum relaxed bounds of the linear function that is evaluated based on the relaxed bounds of its components. We denote the aggregated minimum bound of the linear function by $min[f(\mathbf{v}, \mathbf{d^a})]$, the aggregated maximum bound of the linear by $max[f(\mathbf{v}, \mathbf{d^a})]$ and they are evaluated as follows:

$$min[f(\mathbf{v}, \mathbf{d^a})] = w_1 \cdot lb(v_1) + \cdots + w_n \cdot lb(v_n) + w_{n+1} \cdot lb(d_1^a) + \cdots + w_{m+n} \cdot lb(d_m^a) + c$$

$$max[f(\mathbf{v}, \mathbf{d^a})] = w_1 \cdot ub(v_1) + \cdots + w_n \cdot ub(v_n) + w_{n+1} \cdot ub(d_1^a) + \cdots + w_{m+n} \cdot ub(d_m^a) + c$$

Note that in case any of the lower bounds of variables or parameters take $-\infty$, then $min[f(\mathbf{v}, \mathbf{d^a})]$ is assigned to $-\infty$. Similarly, in case any of the upper bounds of variables or parameter take ∞ then $max[f(\mathbf{v}, \mathbf{d^a})]$ is assigned to ∞ at this action layer.

4.3.1.2 Constant and Non-Constant Bounded Control Parameters

Although the control parameters are defined to take their values from an infinite domain, identifying how their domain bounds react to the changes in the world-state (or when alternating between layers in heuristic) is indispensable to their use. Depending on the

constraints acting upon them, their domains can get diverged or converged or can remain unchanged. The behaviour of the parameter, whose bounds remain unchanged is relatively more predictable than the ones that get diverged (or converged). For instance, a point that is predicted to lie in the domain of a parameter may lie outside of its domain at the next step if the parameter has diverging (or converging) bounds. Whereas the point is guaranteed to remain in its domain if the parameter acts independently (from other variables and parameters) in the heuristic. One can easily detect its behaviour based on the pre- or post-conditions in the domain model: if the parameter is only constrained by constant numbers, its domain will not be affected by the values of other parameter or variables. On the grounds of this relationship, we provide the formal definitions of distinguished behaviours of the control parameters as follows:

Definition 14 (Constant Bounded Control Parameters) *A control parameter* $d_i^a \in \mathbf{d^a}$ *of an action* a *is a constant bounded control parameter if and only if the numeric conditions acting upon* d_i^a *do not include any other control parameters or variables. Each condition must be in the form* $\langle f(\emptyset, d_i^a), comp, c\rangle$.

Definition 15 (Non-Constant Bounded Control Parameters) *A control parameter* $d_i^a \in \mathbf{d^a}$ *of an action* a *is a non-constant bounded control parameter if and only if there is at least one condition acting upon* d_i^a *that includes other control parameters of variables. These conditions are in the form* $\langle f(\mathbf{v}, \mathbf{d^a}), comp, c\rangle$, *where the sizes of* $\mathbf{v}$ *and* $\mathbf{d^a}$ *are non-zeros.*

For instance, the control parameter in the *supply_raw_material* action, `?unit`, is a constant bounded control parameter as the action description (see Figure 4.2) only includes numeric conditions in the form $\langle f(\emptyset, d_i^a), comp, c\rangle$ (e.g. `(at start (<= ?unit 10))`). The control parameter in the *assemble_D* action, `?unit`, is a non-constant bounded control parameter as the action description includes at least one numeric condition in the form $\langle f(\mathbf{v}, \mathbf{d^a}), comp, c\rangle$ (e.g. `(at start (>= (stock ?h) (* ?unit 5)))`).

4.3.2 Producer-Consumer Actions

Each action in a PDDL domain model holds different role based on their standardised syntactic representation when expressing their formal semantics. Each numeric and propositional semantic expressivity option of PDDL versions poses a unique challenge to planning systems. However, the characteristics of actions (in terms of their functionality in the domain) are not directly expressed in the language and this consequently necessitates a detection mechanism to handle them efficiently. For instance, Coles et al. (Coles et al., 2009a) proposed a way to detect *one-shot actions*, which delete their own propositional precondition that cannot be restored, from the domain model. They proposed a method to handle this type of actions delicately in forwards search. Another work that simplifies the planning process by detecting (from PDDL domain model) and merging some actions to form *macro-actions* (Botea et al., 2005). In this thesis, we propose a way to detect actions that hold a producer or a consumer or both characteristics synthesised from the domain model and propose ways to handle them more effectively in the heuristic. We give the formal definitions of such actions as follows.

Let k be a positive integer denoting the occurrence limit of the action $a \in A$ (meaning that how many times an action is estimated to occur at an action layer), and l be the index of the action layer, at which the action a appears in the heuristic. A producer action is defined as follows:

Definition 16 (Producer Action) *An instantiated action* $a_k^l(\mathbf{z})$ *is a* producer *of* $\mathbf{z} \subseteq \mathbf{v}$ *if and only if*

- *it increases each* $v \in \mathbf{z}$*; either with a constant:* $\langle v, \nearrow, c \rangle$*, or a linear function of constant bounded* $\mathbf{d^a}$*:*$\langle v, \nearrow, f(\emptyset, \mathbf{d^a}) \rangle$ *at action layer* l*,*
- *it does not have any numeric condition on any numeric variable in* $\mathbf{z}$*.*

The second bullet point in the definition is critical and it refers that the producer action increases the values of the state variables in $\mathbf{z}$ *u*nconditionally. Thus, we can infer that a producer action can be considered to be applied arbitrarily many times at action layer l (the occurrence limit of a producer $a_k^l(\mathbf{z})$ is always $k = \infty$)

Let $lb(v)$ be a constant denoting to the lower bound of $v \in \mathbf{z}$, where $\mathbf{z} \subseteq \mathbf{v}$. A constant bounded consumer action is defined as follows:

Definition 17 (Constant Bounded Consumer Action) *An instantiated action* $a_k^l(\mathbf{z})$ *is a constant bounded consumer of a vector of state variables* $\mathbf{z}$ *if and only if it has at least a pair of a numeric precondition and a decrease effect acting on all numeric variables* v *in* $\mathbf{z}$*:* $\langle pre_y^n(a),\ eff_y^-(a) \rangle$*, where:*

$$pre_y^n(a) = \langle w \cdot v, \{\geq, >\}, lb(v) + c \rangle$$

$$eff_y^-(a) = \langle v, \searrow, f(\emptyset, \mathbf{d^a}) \rangle$$

This definition has the following important consequence. A constant bounded consumer action monotonically decreases the numeric resource v with a linear function of $\mathbf{d^a}$. Note that the decrease on v may or may not be *strictly monotonic* depending on the aggregated minimum bound of the linear function $f(\emptyset, \mathbf{d^a})$, which is $min[f(\emptyset, \mathbf{d^a})]$. If $min[f(\emptyset, \mathbf{d^a})]$ is equal to zero at a state s, the decrease effect on v would be a *non-decreasing monotonic* effect (or, we can say it is a *fruitless* effect). Choosing this action as the consumer of v consistently during search will eventually lead to a dead-end. If $min[f(\emptyset, \mathbf{d^a})]$ is greater than zero, the decrease on v is *strictly monotonic* and the numeric effect is either a *weakly* or a *strongly* decreasing effect depending on the magnitude of $min[f(\emptyset, \mathbf{d^a})]$. The conceptualisation of this numeric behaviour is vital to this work, as it helps us to determine whether an action should or should not be promoted as helpful during heuristic evaluation.

We define a non-constant bounded consumer as follows:

Definition 18 (Non-constant Bounded Consumer Action)
An instantiated action $a_k^l(\mathbf{z})$ *is a non-constant bounded consumer of* $\mathbf{z} \in \mathbf{v}$ *iff it has at least a pair of a numeric precondition and a decrease effect acting on all numeric variables* v *in* $\mathbf{z}$*:* $\langle pre_y^n(a),\ eff_y^-(a) \rangle$*, where:*

$$pre_y^n(a) = \langle w \cdot v, \{\geq, >\}, f(\emptyset, \mathbf{d^a}) \rangle$$

$$\mathit{eff}_y^-(a) = \langle v, \searrow, f(\emptyset, \mathbf{d^a}) \rangle$$

This definition has two important features, and these are:

1. A non-constant bounded consumer, a, can only be applied if $v \geq f(\emptyset, \mathbf{d^a})$. Thus, the minimum amount that v can take at a state s depends on the aggregated minimum bound of $f(\emptyset, \mathbf{d^a})$, $min[f(\emptyset, \mathbf{d^a})]$. Analogously, the maximum amount that each $d_i^a \in \mathbf{d^a}$ parameter of action a can take relies on the upper and lower bounds of v, $lb(v)$ and $ub(v)$, the aggregated maximum bound of $f(\emptyset, \mathbf{d^a})$, $max[f(\emptyset, \mathbf{d^a})]$, at state s.

2. This action has a monotonically decreasing effect on v with a linear function of $\mathbf{d^a}$. Since the variables in this effect are inter-dependent to each other, it is a self-consuming action, where its effects lead to violation of its conditions during search. This behaviour is analogous to k-shot[1] actions defined in prior work (Coles et al., 2009a)

Having defined the principles of both consumer and producer actions, we now define a far more advanced type that meets the fundamentals of both: *a finite achiever action.*

Let $\mathbf{v_p} \subseteq \mathbf{v}$ denote a vector of state variables that are produced and $\mathbf{v_c} \subseteq \mathbf{v}$ denote a vector of state variables that are consumed by the increase ($\mathrm{eff}_y^+(a)$) and decrease ($\mathrm{eff}_y^-(a)$) effects of the action a, respectively. Then we define the finite achiever action as follows:

Definition 19 (Finite Achiever Action) *Action* $a_k^l(\mathbf{v_p}, \mathbf{v_c})$ *is a* finite achiever *action of* $\mathbf{v_p}$*, where* k *is the occurrence limit at layer* l*, if and only if:*

- *it has at least a producer numeric effect on each* v_p *in* $\mathbf{v_p}$ *in the form:*

$$\mathit{eff}_y^+(a) = \langle v_p, \nearrow, f_p(\emptyset, d_i^a) \rangle$$

- *it has at least a pair of a numeric condition and an effect acting on each consumed numeric variable* v_c *in* $\mathbf{v_c}$*:* $\langle \mathit{pre}_y^n(a),\ \mathit{eff}_y^-(a) \rangle$*, where:*

$$\mathit{pre}_y^n(a) = \langle w_c \cdot v_c - w_d \cdot d_i^a, \{\geq, >\}, c \rangle$$

$$\mathit{eff}_y^-(a) = \langle v_c, \searrow, f_c(\emptyset, d_i^a) + c \rangle$$

- *no other action increases the numeric value of* v_p*.*

We can summarise this definition with the following conclusions:

1. A finite achiever action is both a producer and a non-constant consumer action. It produces each numeric variable v_p with a linear function of $f_p(\emptyset, d_i^a)$ and consumes each numeric variable v_c with a linear function of $f_c(\emptyset, d_i^a)$. Previous work proposed ways to couple actions and add a new inferred action (called *macro-actions*) to the domain to simplify the planning process (Botea et al., 2005; Coles et al., 2007; Gregory et al., 2010).

[1] The notation of k in the work of Coles et al. refers to how many times an action can appear during the search. This contradicts with our definition of k, which we use to describe how many times an action can appear at an action layer in the heuristic graph.

2. As a finite achiever action is a variant of a non-constant bounded consumer action, they both share the following common feature. The numeric condition $\text{pre}_y^n(a)$ and effect $\text{eff}_y^-(a)$ upon variable v_c generates a self-consuming case for such actions.

3. In case any action other than $a_k^l(\mathbf{v_p}, \mathbf{v_c})$, let's say action m, can increase the value of v_p without consuming any numeric resources, then the amount by which v_p can be increased would not be constrained by any other variable, which means the execution of the action m would be the economic choice to achieve v_p. In this case, promoting $a_k^l(\mathbf{v_p}, \mathbf{v_c})$ in the heuristic as the primary achiever of v_p would not be necessary.

4. This action produces each numeric variable v_p to a certain extent. The increasing amount of this action on v_p depends on the available amount of v_c. In this action, the control parameter d_i^a acts as a constrained flow rate between the domains of two variables. In this definition we also facilitate having multiple producer and consumer effects on different produced and consumed variables in the same action. This creates a situation in which multiple resources are simultaneously transformed to other types of resources. We survey this in depth in Section 4.3.3.

It is certainly possible that we can encode many more instances of actions with a producer-consumer behaviour in the planning domain. The ones we define in this work are the most commonly used types in resource transfer with non-constant quantities (with control parameters). A producer replenishes (and a consumer drains) a resource with a non-constant quantity subject to availability of other resources. Furthermore, these definitions not only allow encoding a simple producer-consumer resource transfer, but they also allow encoding cascading resource transfer (a producer of a resource (x) is a consumer of another resource (y), and the producer of (y) can be a consumer of (z), and so on). In this work we address these issues by providing further assistance during heuristic analysis.

4.3.3 Resource Transformation Flow Problem

Each finite achiever action transforms multiple numeric resources to other types. We conceptualise this transformation as a flow problem, called *resource transformation flow*, and define it as follows. Suppose that v is a numeric resource, U is a set of nodes and E is a set of directed edges in a network flow. $\boldsymbol{f}[v, (u_1, u_2)]$ denotes the numeric flow of the resource v on a directed edge $(u_1, u_2) \in E$, where $u_1, u_2 \in U$.

We reduce the resource transformation of a finite achiever to a node, $g \in U$, which has in- and out-flows, $\boldsymbol{f}[v_c, (u_j, g)]$ and $\boldsymbol{f}[v_p, (g, h_k)]$, respectively.

$\boldsymbol{f}[v_c, (u_j, g)]$ denotes the in-flow by which the consumed resource v_c is decreased in j^{th} consumer effect (i.e. $f_c(\emptyset, d_i^a)$) of the action, where it is constrained as follows:

$$min[f_c(\emptyset, d_i^a)] \leq \boldsymbol{f}[v_c, (u_j, g)] \leq max[f_c(\emptyset, d_i^a)]$$

$\boldsymbol{f}[v_p, (g, h_k)]$ denotes the linear out-flow by which the produced resource v_p is increased in k^{th} producer effect (i.e. $f_p(\emptyset, d_i^a)$) of the action, where:

$$min[f_p(\emptyset, d_i^a)] \leq \boldsymbol{f}[v_p, (g, h_k)] \leq max[f_p(\emptyset, d_i^a)]$$

Figure 4.5 illustrates an example resource transformation flow in a finite achiever action where it has three consumer and three producer effects. The red edges represent the in-flow of consumed resources and green edges represent the out-flow of produced resources at planning step g.

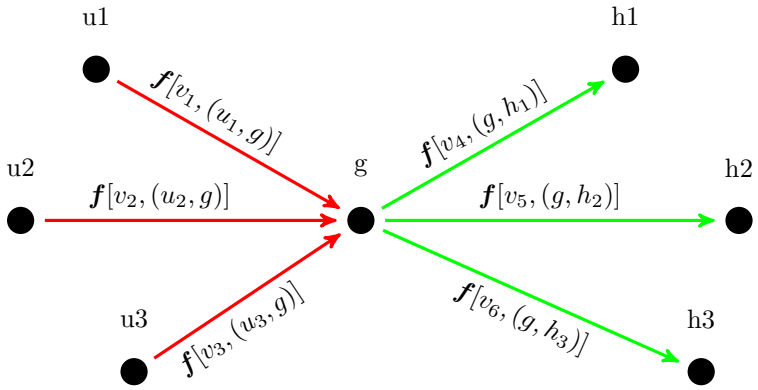


Figure 4.5: Resource transformation flow network representation of the resource transfer in a finite achiever action.

In order to achieve a simultaneous and uninterrupted transformation of resources, it is necessary to decompose the total transformation flow network into individual transformations (the flow from a source to a sink). To achieve this, we evaluate the transformation flow of each consumed resource to its destination produced resource. We consider each combination of producer-consumer effect affected by the same control parameter, as each combination represents a transformation flow and its control parameter behaves as the flow rate. We call each combination of a producer-consumer effect as *constrained resource flow* and we define it as follows:

Definition 20 (Constrained Resource Flow) *Suppose that the action $a_k^l(\mathbf{v_p}, \mathbf{v_c})$ has n consumer numeric effects on $\mathbf{v_c}$ and m producer numeric effects on $\mathbf{v_p}$ with the same control parameter d_i^a, where they are in the form:*

$$eff_y^{+}(a) = \langle v_p, \nearrow, f_p(\emptyset, d_i^a)\rangle$$

$$eff_y^{-}(a) = \langle v_c, \searrow, f_c(\emptyset, d_i^a)\rangle$$

$\boldsymbol{\theta}(a, d_i^a)$ denotes a set of size of $(n \cdot m)$ that contains all possible combinations of constrained resource flows of action a over d_i^a, where each constrained resource flow, denoted as $\theta(a, d_i^a) \in \boldsymbol{\theta}(a, d_i^a)$, is a pair of $eff_y^{+}(a)$ and $eff_y^{-}(a)$.

Considering individual flows from a sink to a source in a network flow is a common practice in network optimisation. In our case, each individual flow (i.e. the constrained resource flow) plays a critical role in finding the bottleneck resource of action a, and consequently helps the heuristic to identify how many times a finite achiever can be executed subject to available quantity of the bottleneck resource. We elaborate on the use of constrained resource flow in Section 4.5.1.

Remark 2 *In a finite achiever action, each constrained resource flow occurs with a single control parameter. This can easily be extended to a multi-variate representation, in which the flow occurs through multiple flow rates (i.e. multiple control parameters). In this case, a new representation of the constrained resource flow could be proposed, say* $\theta_{multi}(a, \mathbf{z^a})$*, where* $\mathbf{z^a} \subseteq \mathbf{d^a}$*. One way to handle this flow would be breaking it into multiple individual flows through the same control parameter, as follows:*

$$\theta_{multi}(a, \mathbf{z^a}) = \{\theta(a, z) \mid \forall z \in \mathbf{z^a}\} \tag{4.1}$$

We restrict the rest of our approach to the flows with a single control parameter for the simplicity of our implementation. We consider extending the approach to the multi-variate representation as a future work.

4.4 Producer-Consumer Patterns

In Section 4.3, we identified the categorisation of various producer-consumer actions and variables based on their pre- and post-conditions and effects in domain models, while the interaction between these actions remained undiscovered. In this section we take a step further identifying the nature of numeric resource transfer (using the producer and consumer actions defined) in temporal planning setting. We define possible sequential orderings of producer and consumer actions, which we call *patterns*. We make use of these patterns during graph expansion of our proposed approach (see Section 4.5).

We identified that the infinity analysis imposes a significant abstraction on the number of occurrence limits of consumer actions, which leads to lack of numeric informedness between actions, and consequently, leads to helpful action distortion. To tackle this problem we establish a numeric predecessor-successor relationship between producer-consumer actions with the patterns we introduce here, and these patterns are: *basic producer-consumer pattern* and *cascading producer-consumer pattern*. These patterns form the backbone of our refined infinity analysis approach.

Suppose that action $a^l_k(v_1, v_2)$ (we assume a^l_k has only one produced and one consumed variable for simplicity) is a finite achiever that produces v_1, and it consumes v_2, the action $c^l_k(v_2)$ is a producer action that produces v_2. In definitions of both patterns we reduce the start-end snapshots of each action to an instantaneous action for simplicity. We present the basic producer-consumer pattern, as follows:

Definition 21 (Basic Producer-Consumer Pattern)

A basic producer-consumer pattern is a chain of partially-ordered actions consisting of a producer, $c^l_k(v_2)$*, and a finite achiever of* v_1*,* $a^l_k(v_1, v_2)$*. For* $l_1 \leq l_2 \leq l_3$*, the following conditions must hold true:*

- $a^{l_1}_{k_1}(v_1, v_2)$ *appears finite times* k_1 *at action layer* l_1
- $c^{l_2}_{k_2}(v_2)$ *appears* $k_2 = \infty$ *times at action layer* l_2
- $a^{l_3}_{k_3}(v_1, v_2)$ *appears (or in case* $k_1 \neq 0$*, it re-occurs)* $k_3 = \infty$ *times at layer* l_3

We can draw the following conclusions from this definition:

1. The finite achiever action appears k_1-times at the earliest possible layer, where the k_1 cannot be infinite. This restriction on k_1 results from the fact that variable, v_2, is monotonically consumed without any replenishment on it.

2. In case the occurrence limit of the consumer is zero ($k_1 = 0$), the finite achiever action does not appear at layer l_1, and consequently, it only appears at layer l_3.

3. Since the producer does not have any numeric resource restrictions (as defined in Definition 16), it can appear infinitely many times at the earliest possible action layer.

4. Both producer and finite achiever actions appear infinitely many times in the end. Thus, the optimistic relaxation on the number of alternating layers in the relaxed plan (i.e. infinity analysis) is preserved.

Basic producer-consumer pattern outlines a relatively simple resource transfer behaviour: If there is enough consumed variable (v_2) available at a state, the planner can consider the finite execution of the finite achiever as helpful. Once v_2 is replenished, the planner can consider applying it again.

Figure 4.6 shows how the actions are partially-ordered with respect to each other in the basic pattern. Action A represents the finite achiever, whereas C represents the producer. The numeric transfer on v_1 and v_2 occurs with decrease ($v \searrow$) and increase ($v \nearrow$) effects. Here, we show numeric preconditions in a simpler form: ($v_2 \geq$). This pattern is only triggered when there is a numeric goal to be satisfied on v_1. The arrows show the steps taken during numeric satisfaction of the goal during solution extraction. We discuss the numeric goal satisfaction in depth in Section 4.5.4.

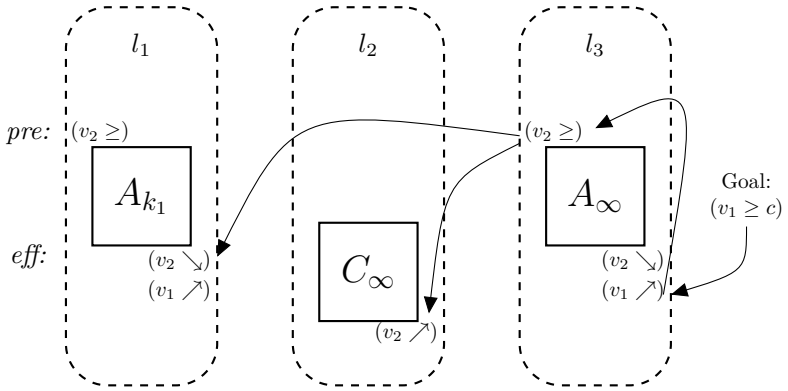


Figure 4.6: The visual representation of the basic producer-consumer pattern.

We can tie this concept to our motivating example as follows. We can observe *at most*[2] 5 different basic producer-consumer patterns (that is indeed equivalent to the number of edges between intermediate products and raw materials in Figure 4.1) in the example. One of the most apparent one is the relationship of the action `assemble_C` (i.e. the finite achiever of

[2]The number can vary depending on the initial stock levels of resources/products and the numeric goal limits.

item C) and the action `(supply_raw_material G)`[3] (i.e. the producer of item g), where the item g is the consumed resource (i.e. v_2) and the item c is the produced resource. Recall that the initial stock levels of these resources are $c = 3$ and $g = 0$. The top-level goal on the item a infers a sub-goal of $c \geq 20$ on item c. As there is initially no item g available, the `assemble_C` can not occur before the `(supply_raw_material G)` appears, as it replenishes g. Overall, this relationship embodies a basic pattern, where the occurrence limit of `assemble_C` at the earliest possible layer is zero (i.e. $k_1 = 0$).

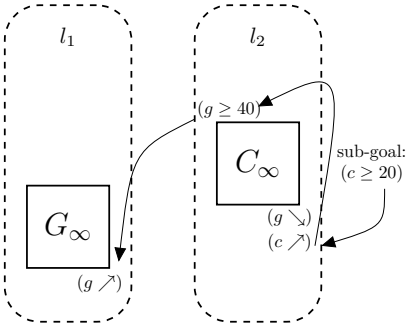


Figure 4.7: Partial ordering of the actions `(assemble_C)` and `(supply_raw_material G)` observed in the procurement example.

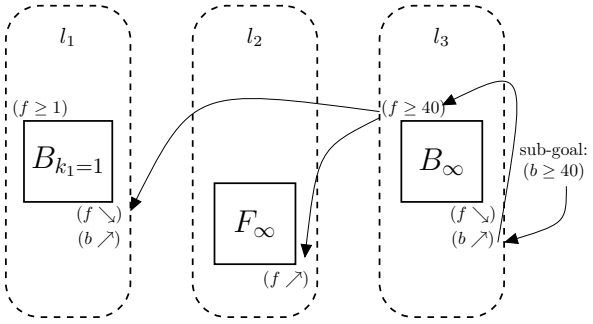


Figure 4.8: Partial ordering of activities, `(assemble_B)` and `(supply_raw_material F)`, observed in the procurement example.

Another basic pattern instance is the relationship between the `assemble_B` (a finite achiever of b) and `(supply_raw_material F)` (the producer of f) actions[4]. The initial stock levels of these resources are given $b = 3$ and $f = 1$. The sub-goal imposed on the item b is $b \geq 40$. Based on the pre- and post-conditions of actions, we can infer that the action `assemble_B` can occur once ($k_1 = 1$) before `(supply_raw_material F)` appears. Figure 4.7 and Figure 4.8 shows the partial ordering of actions in both examples, where we represent `(supply_raw_material G)` and `(supply_raw_material F)` as G and F, respectively.

Suppose that $l_1, l_2, \ldots, l_{2n+1}$ are the action layers in a relaxed planning graph and they are ordered as follows: $l_1 \leq l_2 \cdots \leq l_n \leq l_{n+1} \cdots \leq l_{2n} \leq l_{2n+1}$. We define the cascading producer-consumer pattern, as follows:

Definition 22 (Cascading Producer-Consumer Pattern) *A* cascading producer-consumer pattern $F(v_1) = (a)^l_k(v_1, v_2),\ (a')^l_k(v_2, v_3),\ \ldots,\ (a^{(n)})^l_k(v_{n-1}, v_n)$ *is a chain of partially-ordered actions consisting of finite achievers of* v_1*, and a producer of* v_n*, which is* $c^l_k(v_n)$ *for which the following conditions must hold:*

- $F(v_1)$ *can appear finite times* $(k_1, k_2, \ldots, k_n \neq \infty)$ *at action layers* $l_1, l_2, \ldots, l_n$
- $c^l_k(v_n)$*, the producer, appears* $k_{n+1} = \infty$ *times at action layer* l_{n+1}
- $F(v_1)$ *appears infinite times* $(k_{n+2}, \ldots, k_{2n+1} = \infty)$ *at layers* $l_{n+2}, l_{n+3}, \ldots, l_{2n+1}$.

[3] This action is an instantiated (or grounded) instance of the `(supply_raw_material ?item)` with the object parameter `G`

[4] In the motivating example problem, satisfying numeric goal of $b \geq 40$ triggers slightly more complex pattern, as the initial stock levels of the items d and e are below the required limits. We assume their stocks are in sufficient levels for this small example.

In addition to conclusions we previously stated for basic producer-consumer patterns, we add the following ones for the cascading patterns:

1. A cascading producer-consumer pattern shows the behaviour of a chain of basic producer-consumer patterns: the variable produced in a basic pattern is consumed by another basic pattern, and so on.

2. The finite achievers in the chain are different from each other. It is still possible that, in the planning domain model, some finite achievers show similar producer-consumer behaviour: the variables they produce and consume can be identical. The difference between the two can be the additional numeric or propositional conditions/effects they have. In this case, the planner uses whichever action *achieves its produced variable the earliest* in the relaxed plan. By doing this, we avoid restricting the planner with a total ordering of actions when forming a cascading chain.

3. The necessity of $c_k^l(v_n)$ in the pattern depends on the numeric goal that needs to be satisfied. If the goal indirectly requires replenishment of v_n, then it will be listed in the pattern. This happens when the value of v_n at the current state is sufficient to meet the numeric goal on v_1.

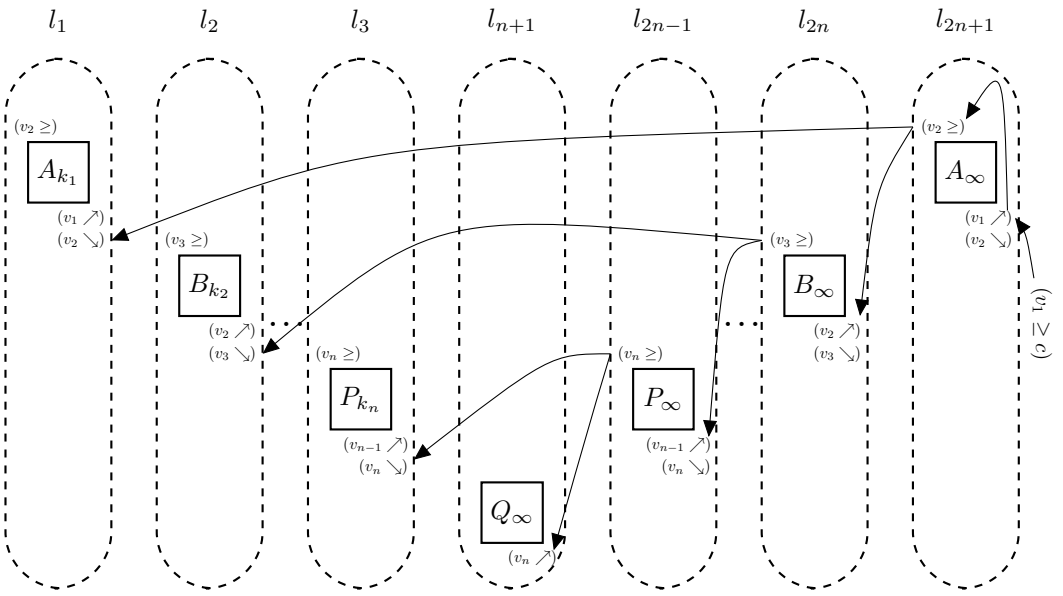


Figure 4.9: The visual representation of the cascading producer-consumer pattern.

Figure 4.9 demonstrates the numeric satisfaction and finite-infinite layering described in the definition above. Here the recursive arrows show the steps taken during solution extraction (when satisfying the numeric goal). Observe that this pattern is only triggered when the heuristic tries to satisfy the numeric goal on v_1, which is supposedly $(v_1 \geq c)$. It is self-evident that multiple use of basic producer-consumer patterns provides a frame for this pattern, in which the basic patterns are linked to each other with a cascade of numeric goals to be satisfied. A simple numeric goal cascades down to multiple goals as follows. The numeric goal $(v_1 \geq c)$ requires $(v_2 \geq)$ sub-goal to be satisfied, while it requires $(v_3 \geq)$

sub-goal to be satisfied, and so on. The heuristic needs to confirm that all these sub-goals can be met to generate a relaxed plan that satisfies the main numeric goal.

In our motivating example, each assemble action decreases the value of their consumed resources, and increases the value of their produced resource by linear functions of `?unit`. Thus, each assemble action in this example is a finite achiever action. Moreover, each grounded instance of the supply_raw_material action is a producer.

We can observe *at most*[5] 4 unique cascading patterns (equivalent to the number of edges between intermediate and final products given in Figure 4.1) in the example. Each cascading pattern has a varying size of partially-ordered actions, and each is individually triggered by a goal (or a sub-goal). For instance, Figure 4.10 shows the cascading pattern triggered by the numeric goal $a \geq 20$. In the figure, all grounded supply_raw_material actions (e.g. `(supply_raw_material G)`, `(supply_raw_material E)` and so on.) are represented as Sr for simplicity. Actions instances in green are the ones that appear finite times, whereas the rest are applied arbitrarily many times. Observe that actions `assemble_C` and `assemble_D` do not have any finite occurrences as the earliest possible layer that they can appear is l_3, at which their consumed resources are already replenished by a set of producer actions (i.e. Sr).

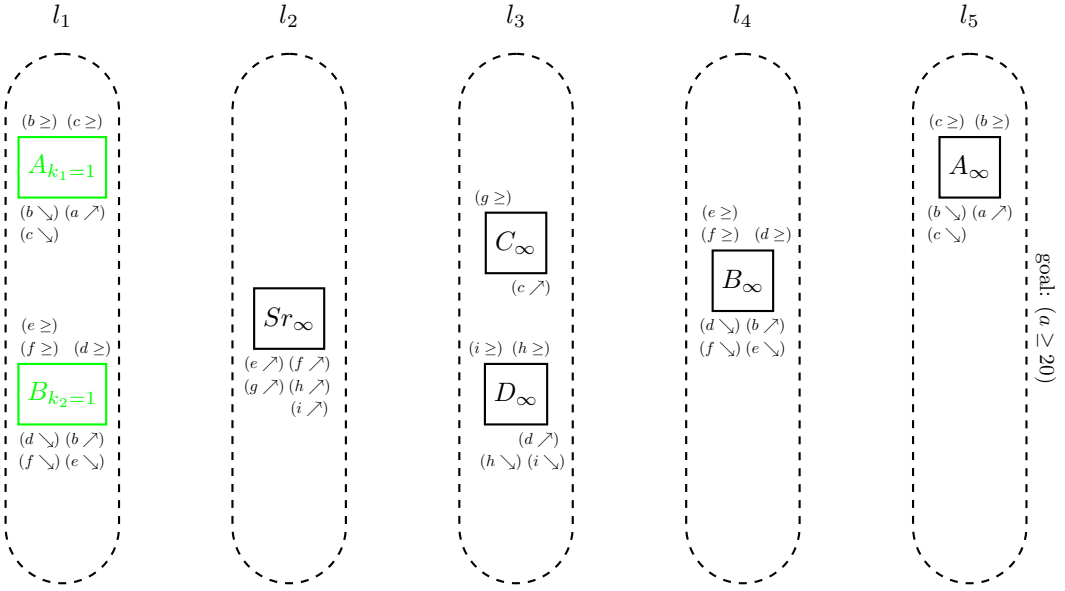


Figure 4.10: The cascading pattern triggered by the goal, $a \geq 20$. Let $l_1 \leq l_2 \leq l_3 \leq l_4 \leq l_5$.

4.5 Extending the TRPG Heuristic

Having identified the producer-consumer relationship between and within actions, we now introduce our approach that improves the basic informedness of the heuristic in producer-consumer problems. Our approach is based on finding a middle ground between the relaxations used in the heuristic of Metric-FF (discretisation of alternating layers) and in POPCORN (infinity analysis). We avoid making use of any optimisation tool within the

[5]The number can vary depending on the initial stock levels and numeric goal limits.

heuristic, as it provides an expensive solution for a heuristic computation. We handle the problem using producer-consumer patterns that we introduced in Section 4.3. In this section we first study how we determine the occurrence limits of actions (k-value). Second, we survey determining the numeric contribution of a non-constant bounded control parameter at an action layer in patterns during graph expansion and extraction. We conclude our contribution by describing the modifications we made in solution extraction of the extended TRPG heuristic used in POPCORN.

4.5.1 Action Occurrence Limits

In planning it is desirable to keep the use of resources and the plan makespan minimal for the benefit of obtaining a good solution quality. When achieving a numeric goal, we expect that the plan does not include a series of fruitless actions that makes insignificant contribution to meet a numeric goal. We examined the dead-end behaviour of the heuristic of POPCORN in our running example (see Figure 4.4), and noticed that a few executions of the assemble_A action can indeed make some numeric contribution towards the goal. The planner should infer that only those occurrences are helpful. This can be achieved in the heuristic by deduction from relaxed numeric bounds. Intuitively, the following question arises at this point: how many times should the planner consider applying such actions without running into a dead-end or sacrificing on the solution quality? In this section we present our approach to estimate the occurrence limit of self-consuming actions to avoid the problems we stated.

Prior work has considered ways to limit the action occurrences in the search. For instance, Coles et al. (Coles et al., 2009a) introduces propositional *one-shot* actions that are only allowed to be applied once in the search space as these actions are *self-violating* (i.e. they remove their propositional conditions, which cannot be achieved by any other action). Similarly, another work (Coles et al., 2013) considers limiting the occurrences of bounded consumer-alike actions (without any control parameters) in the search. We identified that these approaches would significantly affect the applicability of actions with control parameters. Unlike one-shot or k-shot actions, the consumer actions we defined (e.g. a finite achiever) are not self-violating, but they are self-consuming: repetitive executions of our consumer actions do not violate their applicability in the search but it is undesirable to do so. For this reason, we avoid limiting their occurrences in the search, and instead we limit it in the heuristic.

Suppose that Q is an ordered list of actions that have started but not yet finished in search (we refer to them as *open end* actions) and y denotes the start or end snap-actions, $y \in \{\vdash, \dashv\}$. Determining the occurrence limit of a finite achiever action at layer l, $a_k^l(\mathbf{v_p}, \mathbf{v_c})$, depends on the following entities:

1. Each constrained resource flow $\theta(a, d_i^a) \in \boldsymbol{\theta}(a, d_i^a)$ of the action over d_i^a, where the affecting linear functions of its producer and consumer producer effects are in the form:

$$f_p(v_p, d_i^a) = w_p \cdot v_p + w_1 \cdot d_i^a + c$$
$$f_c(v_c, d_i^a) = w_c \cdot v_c + w_2 \cdot d_i^a + c$$

2. The coefficients, w_1 and w_2, of d_i^a in affecting linear functions of both consumer and producer effects,

3. The relaxed upper bound of the consumed variable $v_c \in \mathbf{v_c}$ at layer l, denoted as $ub(v_c, l)$,

4. The relaxed upper bound of the control parameter of the action, $ub(d_i^a)$.

Algorithm 2: Determining occurrence limits

Data : Q - open end actions, l - current action layer, toRevisit - revisit set
Result: $k(a_y)$ - occurrence limit of the snap-action a_y

```
1  Ω - local flow,  k(a_y) ← ∞;
2  if a_y is a snap-action of a finite achiever (a) then
3      if (y = ⊢) ∧ a_⊢ ∉ Q then
4          foreach constrained resource flow of a, θ(a, d_i^a) ∈ θ(a, d_i^a) do
5              Ω ← 0, retrieve w_1 and w_2 from θ(a, d_i^a) ;
6              ratio ← (w_1/w_2);
7              if ub(v_c, l) = ∞ then
8                  Ω ← ∞, return k(a_y) ← ∞;
9              else
10                 Ω ← ub(v_c, l)× ratio;
11             if (Ω ≠ −∞) ∧ (Ω ≠ ∞) then
12                 if k(a_⊢) > ⌈Ω/ub(d_i^a)⌉ then
13                     k(a_⊢) ← ⌈Ω/ub(d_i^a)⌉;
14                     insert a_⊢ into toRevisit;
15     else if (y = ⊣) ∧ a_⊣.op ∉ Q then
16         k(a_⊣) ← k(a_⊢);
17     else
18         k(a_y) ← 1;
19 return k(a_y);
```

Using these entities Algorithm 2 describes how we determine the occurrence limits of actions. We call this algorithm at each action layer during graph expansion. Suppose that a_y denotes the start and end snap-actions of a durative action a. We initially set the occurrence limits of all snap-actions to ∞ (line 1), and this remains unchanged unless the action is a finite achiever. We update the k value for all finite achiever actions in Line 2. In case a_y is a start snap-action but the action has not yet started in search, we compute its k value based on its resource flow, upper bounds of consumed variables and parameters in line 3. In case a_y is an end snap-action but its start has not yet appeared in search, then we assign the k value of its start snap-action to its end (line 15). This assignment is necessary to satisfy one of the main principles of TRPG heuristic: *the end snap-action can only be applied if its start appears before it.* In case a_y is an open end action, then we assume that it can be applied only once (line 17).

We examine every pair of a consumer and a producer effects that have the same control parameter (i.e. the constrained resource flow, $\theta(a, d_i^a)$), because the resource transfer between their variables, v_p and v_c, are inter-dependent through d_i^a in line 4. We surveyed the concept of the constrained resource flow $\theta(a, d_i^a)$ in depth in Section 4.3.3. We initialise

the local flow variable (Ω) for each constrained resource flow to zero (line 5), and evaluate the local flow of consumed variable v_c to produce v_p based on relaxed upper bound of the consumed variable. In case the bound is ∞ (which means v_c has been replenished in previous layers), k remains ∞ (line 8); otherwise k value is computed using the coefficients of d_i^a and the upper bound of v_c (line 10). In case the local flow of the resource in $\theta(a, d_i^a)$ is finite (line 11), we assign the smallest local flow *per flow rate* (that is d_i^a) as the k value of a_y (line 13). Observe that the value assigned is always a finite number. We then record this snap-action to be *revisited* in further action layers (line 14). If the $k(a_y)$ leaves the algorithm as ∞ (as in line 8), this snap-action does not need to be revisited again in this heuristic.

It is important that the relaxation considers the most optimistic scenario to check whether all goals are ever achievable. To maintain this, we use the infinity analysis of TRPG, which is formed based on this idea: *if an action is applicable at a layer, it is considered to be applicable infinitely many times.* We modify this for our case as: *if a finite achiever action is applicable at a layer, it is applicable finite times until at a layer all of its resources are replenished; then it becomes applicable infinitely many times.* This principle ensures that all finitely applied actions are applied infinitely many times in the end.

As previously mentioned, we record the actions, which needs to re-appear in the graph, in a set called *toRevisit*. In every further action layer, we check whether the numeric resources of these actions are replenished (by simply checking relaxed upper bound of each resource at that layer). If the upper bounds of all has already reached to ∞ (line 8) at a layer, we re-apply (or revisit) this action (and all of its effects) infinitely many times at this layer. We present the details of exploiting revisiting mechanism in depth in Section 4.5.3. Consequently, this leads us to complete the basic producer-consumer pattern (see Definition 21) for the resource transfer between v_p and v_c. We follow the same steps for more complex numeric transfer problems like cascading patterns.

4.5.2 Numeric Contribution of Non-Constant Bounded Parameters

A finite achiever action is a variant of a non-constant bounded consumer action, where the upper bound of the affecting control parameter depends on the values of other variables. Recall that an LP was used (only once per parameter) in the heuristic of POPCORN to compute the relaxed bounds of the non-constant bounded control parameters, which is deemed insufficient (as it was only used once). Instead of using an LP, we propose to handle this issue with a realistic and efficient estimate of these parameters with a simple arithmetic comparison operation subject to their limiting variables. We call the resulting estimate as the *numeric contribution* of the corresponding parameter.

Let w be a real-valued constant, where it is predefined in the domain model. We define the limiting variable as follows:

Definition 23 (Limiting Variable) *A limiting state variable of a control parameter d_i^a is a variable $v \in \mathbf{v}$ that constrains the bounds of d_i^a with a numeric condition, $pre_y^n(a)$, where it is in the form:*

$$pre_y^n(a) = \langle w \cdot v, \{\geq, >\}, f(\emptyset, d_i^a)\rangle$$

Before we start presenting the method, recall the differences between the true and the relaxed bounds of a numeric variable. The true bound (of a variable at a state) refers to the numerically consistent range of values that the variable can take (usually checked by a mathematical solver, if necessary). Any value lies outside the range makes that state *numerically inconsistent*. In producer-consumer actions, the true lower bound of the bottleneck resource determines whether the repetitive execution of the action helps the planner to satisfy the numeric goal during search. We studied the relationship between the bottleneck resource b, and the numeric goal on a in Section 4.2. The relaxed bounds of a variable refers to an estimate of the bounds of a variable in a relaxation-based heuristic. In planning graph-based heuristics, the relaxed bounds get diverged (or remain unchanged) as the heuristic alternates between fact and action layers.

Algorithm 3: Numeric contribution of d_i^a in eff(a) of the snap-action a_y

Data : $a_k^{l_2}(\mathbf{v_p}, \mathbf{v_c})$, d_i^a in eff(a), $k(a_y)$, l_1 - previous fact layer, l_2 - current action layer
Result: $\mathcal{C}$ - total contribution of d_i^a

```
1  φ - guaranteed contribution, β - possible contribution;
2  C ← ∞,   β ← ∞,   φ ← ∞;
3  forall pre(a) in the form v ≥ w_d · d_i^a + c do
4      if 0 ≤ lb(v, l_1) < φ then
5          φ ← lb(v, l_1)/w_d;
6      if 0 ≤ ub(v, l_1) − ub(d_i^a, l_1) < β then
7          β ← [ub(v, l_1) − ub(d_i^a, l_1)]/w_d;
8  if k(a_y) ≠ ∞ then
9      C ← (φ + β)/k(a_y);
10 return C;
```

Suppose that $l_1 \leq l_2$, and $ub(v, l)$ and $lb(v, l)$ denote the upper and lower relaxed bounds of v at layer l, respectively. Algorithm 3 computes the numeric contribution of a non-constant control parameter in a numeric effect of $a_k^{l_2}(\mathbf{v_p}, \mathbf{v_c})$, eff($a$). In this algorithm we split the numeric contribution in two: *guaranteed* and *possible* contributions. Initially both contributions are set to ∞ in line 2. We iterate through all numeric preconditions of a, $pre(a)$, to find the limiting variable (in line 3-7). In line 4-5, whichever limiting variable over d_i^a has the least relaxed lower bound at the current action layer determines the *guaranteed numeric contribution* of d_i^a. In similar fashion, whichever limiting variable over d_i^a has the least relative difference between its relaxed upper bound and the relaxed upper bound of d_i^a (i.e. $ub(v, l_1) - ub(d_i^a, l_1)$) determines the *possible numeric contribution* of d_i^a (line 6-7). We refer to this relative difference as the *slack of a parameter*, and it has the following components:

- The slack of d_i^a always takes a non-negative value, as $pre(a)$ ensures that the least value $ub(v, l_1)$ can take is equal to $ub(d_i^a, l_1)$.

- In case the slack of d_i^a is not equal to zero, the upper bound of the limiting variable of d_i^a must be increased in previous action layers. Then, the slack gives the amount by which the limiting variable is increased.

In the algorithm, the constant w_d denotes the coefficient of d_i^a. We use this constant in this algorithm to evaluate the numeric contribution *per execution* of the action. In case the occurrence limit of the action is not ∞, then the total contribution of d_i^a in an effect at the action layer l is the ratio of the sum of possible and guaranteed contributions and the occurrence limit of this action (line 9). In case $k(a_y) = \infty$, then the total contribution remains ∞ as all of its limiting variables are replenished in earlier action layers.

4.5.3 Building the Temporal RPG Using Refined Infinity Analysis

We presented computing the occurrence limits of actions within the heuristic and the numeric contribution of a non-constant bounded parameter in previous sections. We now describe how all of these components are integrated in graph expansion phase of the Temporal RPG (TRPG) exploiting refined infinity analysis. We employ the most up-to-date TRPG graph expansion algorithm reported in the work of Coles et al. (Coles et al., 2008a). Our modifications respect the following temporal principles of the TRPG:

- Reasoning with start-end semantics of PDDL 2.1,
- In order to apply the end of an action, its start should appear first,
- The end of an action can only appear after a sufficient time has passed.

These principles aim to provide intelligible guidance for domain models obeying the start-end semantics of PDDL 2.1 language. The TRPG approach used in Sapa planner (Do and Kambhampati, 2014) compresses a durative action into a temporally-extended action that fails to satisfy the first principle given above. The TRPG heuristic we use (of CRIKEY planner and its successors) splits a durative action into a start and an end snap-action respecting all of the principles given above. It replaces the layer indices of the RPG with time-stamped indices to provide more informative temporal relaxation.

Exploiting the refined infinity approach in the TRPG framework requires an extension to its main principles, and these are:

1. In our approach, a snap-action can appear in multiple layers in the same graph. Therefore, the heuristic needs to make sure that the start of each end snap-action instance is included in the same graph.
2. In the existing mechanism of the TRPG, it is not possible to associate the occurrence limits of snap-action instances with the action layer at which they appear.

We can give an example problem that would arise from ignoring the first requirement of our approach. Suppose that $l_1 \leq l_2 \leq l_3$ are action layers. The start snap-action of a appears at layer l_1 for $k(a_\vdash) = 1$, and the end snap-action of a at layer l_2 for $k(a_\dashv) = 1$. Assume that the end snap-action is recorded to be revisited in further action layers, and it re-appears at layer l_3 for $k(a_\dashv) = \infty$. In this case, none of the existing principles of the TRPG ensures that another start snap-action is needed (as a start snap-action instance has already appeared at layer l_1). The heuristic can *incorrectly* reach to a dead-end. To avoid this misrepresentation, we add the following principle for our case:

Let al_t denote the t^{th} action layer in a relaxed planning graph $\mathcal{R} = \langle fl_{0..n}, al_{0..n} \rangle$.

– The net occurrence limit of the start snap-action of a is equal to the net occurrence limit of the end snap-action of a:

$$\sum_{t=0}^{n}[k(a_{\vdash}) - k(a_{\dashv})] \text{ at } al_t = 0$$

An action layer is defined to be a set of snap-actions whose preconditions appeared in the previous fact layer, however it should also include how many times each snap-action can be applied at this layer. We replace the action layer representation of the heuristic with a set of pairs that maps the snap-action to its occurrence limit, as follows:

Let r denote an entry of the action layer al_t, and a_y denote a snap-action. Each entry is in the form:

$$r = \langle a_y, k(a_y)\rangle, \text{ such that } r \in al_t$$

Let fl and al denote fact and action layers, respectively, and $earliest(a)$ denote the earliest time-stamp at which the end of action a can appear, ep denote the endpoint of an action. Algorithm 4 outlines the modifications we made to the graph expansion of the TRPG heuristic first used in CRIKEY (Coles et al., 2008a). The algorithm is extended so that it can take a $C-$state as an input and it can reason with the refined infinity analysis.

The algorithm takes the propositional and numeric facts at state S as an input to initialise the initial fact layer (line 4). If we are evaluating a state at which an action has already started but not yet finished, the earliest possible time-stamp of the end of this action becomes 0, or otherwise it is ∞ (line 2). At line 9, if the end of action a becomes applicable and sufficient time has passed for the end of a, we obtain the occurrence limit of $a_{\dashv}$ and record the action and k-limit pair to the current action layer. At line 13, we check whether any actions can be revisited as a part of refined infinity analysis. If the occurrence limit of the action to be revisited is computed as ∞, then we record this action to the current action layer. At line 18 for each numeric effect with a non-constant parameter of a newly added end snap-action, we compute the numeric contribution of d_i^a and re-evaluate its relaxed bounds. The updated relaxed bounds of d_i^a are then used when we evaluate the numeric effect that updates the numeric facts of the next fact layer. At line 22 we add the start snap-action of the newly added end snap-action to current action layer if all of its conditions are satisfied. At line 26, similar to line 18, we compute the numeric contribution of non-constant bounded parameters to evaluate the numeric effect to be added to next fact layer. The earliest time that the newly added end snap-action is updated based on duration constraints of a (line 30). Here $minDur(a)$ denotes the minimum duration limit of the action a. The planning graph expands until there is no end snap-action to be added *and* no new propositional/numeric fact becomes true (line 31) in the current relaxed planning graph. The algorithm returns an ordered list of fact and action layers to be used in relaxed plan solution extraction phase.

In short, the modifications we made to the graph expansion algorithm of the TRPG (which is Algorithm 4) is mostly on action layers. We altered the action layer representation to include occurrence limits associated with their snap-actions. We compute the occurrence limit of snap-actions every time they are added to the planning graph. We re-evaluate the relaxed bounds of non-constant control parameters based on their estimated numeric contribution whenever they are encountered in the graph. Then the updated relaxed bounds

Algorithm 4: Building the TRPG Using refined infinity analysis

Data : $S = \langle F, V, B, Q, P, C, T, D \rangle$ - the state to be evaluated
Result: $\mathcal{R} = \langle fl_{0..n}, al_{0..n} \rangle$ - a relaxed planning graph

$fl_0 \leftarrow [F, V]$;
$t \leftarrow 0$;
foreach $a_\dashv$ **do**
  if $a \in Q$ **then**
    $earliest(a) \leftarrow 0$;
  else
    $earliest(a) \leftarrow \infty$;
while $t < \infty$ **do**
  $fl_{t+\epsilon} \leftarrow fl_t$;
  if $pre(a_\dashv) \subseteq fl_t \wedge earliest(a) \leq t$ **then**
    retrieve $k(a_\dashv)$ from Algorithm 2;
    $al_t \leftarrow al_t \cup \langle a_\dashv, k(a_\dashv) \rangle$;
  foreach *snap-action* $b_y \in toRevisit$ **do**
    retrieve $k(b_y)$ from Algorithm 2;
    if $k(b_y) = \infty$ **then**
      $al_t \leftarrow al_t \cup \langle b_y, k(b_y) \rangle$;
  foreach *new* $a_\dashv \in al_t$ **do**
    foreach *numeric effect eff*(d_i^a) *of* $a_\dashv$ **do**
      retrieve $\mathcal{C}$ from Algorithm 3;
      re-evaluate the relaxed bounds of d_i^a and the values $min[f(\emptyset, d_i^a)]$, $max[f(\emptyset, d_i^a)]$ of eff(d_i^a) with respect to $\mathcal{C}$;
    $fl_{t+\epsilon} \leftarrow fl_{t+\epsilon} \cup$ eff $^+(a_\dashv)$;
    if $pre(a_\vdash) \subseteq fl_t$ **then**
      $k(a_\vdash) \leftarrow k(a_\dashv)$;
      $al_t \leftarrow al_t \cup \langle a_\vdash, k(a_\vdash) \rangle$;
  foreach *new* $a_\vdash \in al_t$ **do**
    foreach *numeric effect eff*(d_i^a) *of* $a_\vdash$ **do**
      retrieve $\mathcal{C}$ from Algorithm 3;
      re-evaluate the relaxed bounds of d_i^a and the values $min[f(\emptyset, d_i^a)]$, $max[f(\emptyset, d_i^a)]$ of eff(d_i^a) with respect to $\mathcal{C}$;
    $fl_{t+\epsilon} \leftarrow fl_{t+\epsilon} \cup$ eff $^+(a_\vdash)$;
    $earliest(a) = min[earliest(a), t + minDur(a)]$;
  if $fl_t \subset fl_{t+\epsilon}$ **then**
    $t \leftarrow t + \epsilon$;
  else if $pre(a_\dashv) \subseteq fl_t \wedge earliest(a) > t$ **then**
    $t \leftarrow min[ep]$;
  else
    break;
return $\mathcal{R} = \langle fl_{0..n}, al_{0..n} \rangle$

are used at the numeric effect in which they appear. We ensure that the occurrence limit of start-end snap-actions match to respect all the principles of the TRPG. Last, we check the revisit queue whether any of the snap-actions become infinitely applicable, and if so, they are re-added to the planning graph at this action layer.

The main objective of constructing a planning graph is to check whether all goals become reachable in the most optimistic scenario where we relax the delete effects of actions and their occurrence limits. For this reason, the graph expansion is sometimes called as *reachability analysis*. In addition, the graph expansion phase provides all required details for solution extraction. The graph expansion terminates when all propositional and numeric goals are deemed satisfied in the graph, and the graph includes all the end snap-actions of the currently executing actions. Otherwise, the heuristic reaches to a dead-end. On termination, a relaxed plan is extracted by regressing through the graph starting from the latest layer. We provide the details of solution extraction in Section 4.5.4.

Algorithm 5: Adding the sub-goals of recently added achievers to the priority queue (used in Algorithm 4)

Data : $\mathcal{R} = \langle fl_{0..n}, al_{0..n} \rangle$ - a relaxed planning graph,
a - action added to the relaxed plan, q - the sub-goal queue

```
1 foreach propositional precondition pre of a do
2     t ← the timestamp at which it pre first appears;
3     if t > 0 then
4         q[t].prop ← q[t].prop ∪ pre;
5 foreach numeric precondition pre^n of a do
6     t ← the timestamp at which it pre^n first holds;
7     if t > 0 then
8         q[t].num ← q[t].num ∪ pre^n;
```

4.5.4 Solution Extraction

Having described the graph expansion of the TRPG using refined infinity analysis, we now present the solution extraction process used in our approach in this section. The solution extraction takes the recently built relaxed planning graph of the state S, i.e. $\mathcal{R} = \langle fl_{0..n}, al_{0..n} \rangle$, and all the high-level goals pre-defined by the modeller as the input. It returns a list of helpful actions that are used guiding the search algorithm, a heuristic distance to reach all goals and the details of the relaxed solution plan. The solution extraction is done by regressing through the relaxed graph to satisfy the high-level goals and sub-goals of the problem recorded in a priority queue. For each goal in the queue, its achiever action is added to the relaxed plan and its conditions are added to the list to be satisfied at an earlier layer.

Our extension to the solution extraction is mostly done on numeric goal satisfaction process. Algorithm 6 provides the details of the process built on top of the solution extraction phase of Metric-FF (full detail of the algorithm is provided by Coles et al. (Coles et al., 2013)). In this algorithm, we abstract out start-end and compression safe reasoning of the TRPG for simplicity. We also assume that all numeric goals are converted into linear normal form. Before regressing through the graph, the propositional and numeric goals specified

Algorithm 6: Solution extraction of the TRPG using refined infinity analysis

Data : $\mathcal{R} = \langle fl_{0..n}, al_{0..n} \rangle$, $G(F)$ / $G(V)$ - propositional/numeric goals
Result: $\mathcal{P}$ - the relaxed plan, H - helpful actions, h - a heuristic value

$H \leftarrow \emptyset$, $\mathcal{P} \leftarrow \emptyset$, $h \leftarrow 0$, $q \leftarrow$ priority queue of goal layers;
foreach $f \in G(F)$ **do**
  $t \leftarrow$ the timestamp at which f first appears;
  $q[t].prop \leftarrow q[t].prop \cup f$;
foreach $n \in G(V)$ **do**
  $t \leftarrow$ the timestamp at which n first holds;
  $q[t].num \leftarrow q[t].num \cup n$;
while $q \neq \emptyset$ **do**
  take one from the end of q, and assign to $(t, \langle prop, num \rangle)$;
  foreach $f \in prop$ **do**
    $a_y \leftarrow$ an achiever snap-action for f;
    $h \leftarrow h + 1, \quad \mathcal{P}(t) \leftarrow \mathcal{P}(t) \cup \langle a_y, 1 \rangle$;
    if $t = 0$ **then** add a_y to H;
    call Algorithm 5 with $\mathcal{R}$, q, a; remove add effects of a_y from $prop$;
  foreach $(v \geq c) \in num$ **do**
    if *a snap-action* $a_y \in al(t)$ *has numeric effect* $(v \nearrow e) \vee (v \nearrow f(\emptyset, d_c^a))$, *where* $e \cdot k(a_y) \geq c, \quad ub(d_c^a) \cdot k(a_y) \geq c$ **then**
      $applyTimes \leftarrow \lceil [c - ub(v, prev(t))] / max[f(\emptyset, d_c^a))] \rceil$;
      $h \leftarrow h + applyTimes, \quad \mathcal{P}(t) \leftarrow \mathcal{P}(t) \cup \langle a_y, applyTimes) \rangle$;
      if $t = 0$ **then** add a_y to H;
      remove the goal $(v \geq c)$ from num;
      call Algorithm 5 with $\mathcal{R}$, q, a;
    if *a snap-action* $a_y \in al(t)$ *has numeric effect* $(v \nearrow e) \vee (v \nearrow f(\emptyset, d_c^a))$, *where* $e \cdot k(a_y) < c, \quad ub(d_c^a) \cdot k(a_y) < c$ **then**
      while $ub(v, prev(t)) < c$ **do**
        $a_y \leftarrow$ new increaser of v;
        $applyTimes \leftarrow \lceil [c - ub(v, prev(t))] / max[f(\emptyset, d_c^a))] \rceil$;
        $h \leftarrow h + applyTimes, \quad \mathcal{P}(t) \leftarrow \mathcal{P}(t) \cup \langle a_y, applyTimes \rangle$;
        if $t = 0$ **then** add a_y to H;
        decrease c by the numeric increase of a_y on v;
        if $t > 0$ **then** add $(v \geq c)$ to $q[prev(t)].num$;
        call Algorithm 5 with $\mathcal{R}$, q, a;
    if *a snap-action* $a_y \in al(t)$ *has* $eff(d_n^a) = (v \nearrow f(\emptyset, d_n^a))$ **then**
      while $ub(v, prev(t)) < c$ **do**
        retrieve $\mathcal{N}$ from Algorithm 7 with $\text{eff}(d_n^a)$, $\mathcal{R}$, a_y;
        $applyTimes \leftarrow \lceil c - \mathcal{N} / max[f(\emptyset, d_n^a))] \rceil$;
        $h \leftarrow h + applyTimes, \quad \mathcal{P}(t) \leftarrow \mathcal{P}(t) \cup \langle a_y, applyTimes \rangle$;
        if $t = 0$ **then** add a_y to H;
        decrease c by $(c - \mathcal{N})$;
        if $t > 0$ **then** add $(v \geq c)$ to $q[prev(t)].num$;
        call Algorithm 5 with $\mathcal{R}$, q, a;

Algorithm 7: Regression function to find total numeric contribution of a in previous layers

Data : $\mathcal{R} = \langle fl_{0..n}, al_{0..n} \rangle$ - a relaxed planning graph, a_y - a snap-action to check its finite instances, eff(d_n^a) - the increaser effect, t - current layer
Result: $\mathcal{N}$ - total numeric contribution of a in previous layers of $\mathcal{R}$

```
1  N ← 0;
2  if action a is a finite achiever action then
3      while t > 0 do
4          t ← prev(t);
5          if a_y is found in al(t) then
6              retrieve C from Algorithm 3 with a, t, eff(d_n^a), k(a_y);
7              N ← N + C;
8  else
9      N ← ∞;
10 return N;
```

in the problem file are recorded in the priority queue q in line 2 and 5, respectively, at the timestamps they first appear. The regression of satisfying goals in the graph occurs in line 8. The achiever of each propositional goal in the queue is added to the relaxed plan, and the heuristic value is incremented by one (as it indicates the required number of action instances in relaxed plan that achieves f) in line 10. All propositional add effects of the achiever action is removed from the propositional goals queue as they are also achieved by the same action. The process up to this point is identical to the original TRPG extraction.

We consider the achiever actions of numeric goals in a relatively different way in line 15. Let c and e denote real-valued constants, d_n^a denote a non-constant bounded control parameter, d_c^a denote a constant bounded control parameter of an action a, $prev(t)$ denote the timestamp of the previous layer of t, $succ(t)$ denote the timestamp of the successor layer of t. In case an action has a numeric effect that $k(a_y)$ times execution of it can achieve the numeric goal ($v \geq c$), the achiever is added to the relaxed plan as usual (line 16). In case the $k(a_y)$ times execution of the action is not adequate to achieve the goal, then other achievers of v from earlier layers are added to the relaxed plan *until* the residual value of the goal ($v \geq c$) is small enough to be reachable in earlier fact layers (line 22). Note that, for both cases, the action achieves the goal either by a constant or a linear function of a constant bounded control parameter.

In case the action achieves the numeric goal by a linear function of a non-constant bounded control parameter, then we can trigger our producer-consumer inference. This case is shown in line 31. We obtain the total numeric contribution of the snap-action a_y towards v in previous layers, which is denoted as $\mathcal{N}$ (line 33) (i.e. we assume that we can increase v by $\mathcal{N}$ in previous layers). The remainder value of the goal, $c - \mathcal{N}$, is considered to be achieved at the current action layer. We add the residual value as a sub-goal to be satisfied in previous layers (line 38). The sub-goaling continues as the goal is small enough to be satisfied in earlier fact layers.

All actions in the relaxed plan considered to be helpful if they appear in the earliest action layer (lines 13, 19, 27, 36).

4.5.4.1 Differences Between k-value and $applyTimes$ of an Action

Observe that the occurrence limit of an action is not used as the number of times a_y appears in $\mathcal{P}$, but there is somehow a surrogate, called $applyTimes$. The difference between the two defines the clear-cut distinction between the graph expansion and solution extraction. The occurrence limit of an action reports how many times an action can be applied, in the most optimistic case, to check whether all goals ever become reachable. Whereas $applyTimes$ reports how many instances of an action are required to appear in the relaxed solution plan so that it is adequate to reach all goals. The occurrence limit (k) and $applyTimes$ of an action differs from each other based on the following qualities:

- The k-limit is only used to check the reachability of the goals of the problem, whereas $applyTimes$ provides detail about approximately how far we are away from reaching the goal,
- The occurrence limit can not be less than $applyTimes$,

Note that $applyTimes$ is an estimated limit (but a better indicator to the relaxed solution than k) to the required number of action instances in the relaxed plan. We compute it using the aggregated relaxed maximum of the affecting linear function of the numeric effect (i.e. $max[f(\emptyset, d_c^a))]$ and $max[f(\emptyset, d_n^a))]$) and the amount that is achievable by the action instance at the current action layer (i.e. c minus achievable amount in previous layers).

4.5.4.2 The Relaxed Plan

In Section 4.4, we have identified that our running example embodies 5 basic patterns (between intermediate products and raw materials) and 4 cascading patterns (between intermediate and final products), which are individually triggered by different numeric goals. For instance, if the numeric goal was to achieve $c \geq 30$, we would only see a basic pattern (similar to the one shown in Figure 4.7) triggered. Or, if the only numeric goal was $b \geq 50$, we would see a pattern similar to the one in Figure 4.8. The original goal of the problem (i.e. $a \geq 20$), however, is far more complex than these goals and it triggers the cascading pattern shown in Figure 4.10. One can be concerned about the significance of these patterns in terms of heuristic evaluation. The actions captured in partially-ordered patterns are the ones that must be included in the relaxed plan (also in the solution plan), as each includes *numeric landmarks* to be achieved when satisfying the top-level goal(s) of the problem. POPCORN implemented with refined infinity analysis generated the relaxed solution plan (i.e. $\mathcal{P}$) for the problem (see Appendix A) using the cascading pattern shown in Figure 4.10 and the solution extraction algorithm. The timeline of the relaxed plan is shown in Figure 4.11. In the relaxed plan, the time of each action is the earliest time that each can appear in the planning graph. It becomes apparent in the timeline that the partial-ordering enforced by the patterns does not restrict the executions of actions concurrently.

4.6 Evaluation

In this section we present a detailed assessment of the refined infinity analysis by comparing it to the heuristic of POPCORN and cqScotty. To evaluate our approach we conducted

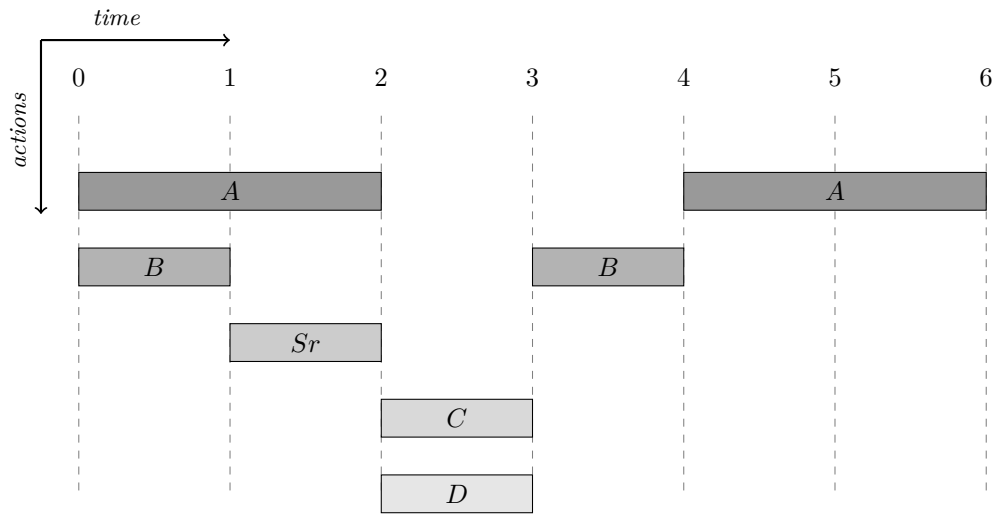


Figure 4.11: Timeline of the relaxed plan of our motivating example generated by POPCORN using refined infinity analysis.

an ablation study on POPCORN and then benchmark it against cqScotty planner on a new expressive domain. We implemented our approach (i.e. refined infinity analysis) on POPCORN, called *POPCORN-R*, where R stands for Refined infinity analysis. All planning systems (including cqScotty) use enforced hill climbing (EHC) followed by best-first search (BFS). The planner resorts to BFS if the EHC fails. The stopping criterion on the search is either reaching a goal state or the planner exceeds the runtime limit (timeout was set to 30 minutes) or it runs out of memory (4GB allocated memory).

In the ablation study, we evaluate the planners POPCORN and POPCORN-R on a set of benchmark domains[6] using a single core 3.4 GHz machine with 16 GB RAM (4GB allocated memory). Since we were not able to run cqScotty or Scotty in benchmark domains used in the ablation study (with POPCORN), we modified one of the benchmark domains of Scotty as we were curious to see how our approach compares with Scotty in their own terms. The domains used in the ablation study include discrete numeric change with control parameters, which is currently not handled by any versions of Scotty. This has been confirmed by direct communication with the authors of this system. Note that the control parameters in these domains are independent from the duration of the action (i.e. the withdrawal amount from an ATM). This feature is not yet supported by Scotty. We summarise the domains as follows.

4.6.1 Evaluation Domains

Before we present the evaluation results, we introduce the domain models used in our tests. We talk about why we choose using these domains and how the problem instances of each domain model are generated. We are not only concerned about evaluating the improvements of using refined infinity analysis on solvability and scalability but also checking whether it is compatible to other planning components. Thus, we considered the following conditions when choosing or developing the domain models:

[6]Benchmarks domain/problems, random problem generators, and obtained result details are available at *bitbucket.org/p3540/aaai*

- Each action can have mixed propositional-numeric, only numeric or only propositional causal links.
- Each action can have a varied number of typed object parameters. Note that the number of typed object parameters of each action controls the number of instantiations of that action.
- Each action can have a varied number of control parameters.
- Some planning problems exhibit required concurrency. Thus, we allow some actions to be executed simultaneously.

We use a set of problem instances for each domain model to test the solvability and scalability of our approach. Each problem instance (that are randomly generated) carries one or a combination of the following qualities:

- We increase the number of objects each type can have in order to make the problems more challenging to solve: this change increases the number of instantiated actions and leads to a wider search space.
- The number of propositional/numeric goal conditions set to be achieved are increased.
- The numeric goals are either simple enough so that the planner does not need to consider replenishing any consumed resources, or moderately challenging so that it requires a few replenishing action instances, or extremely challenging so that it requires many replenishing action instances by cascading sub-goal relationship.
- The numeric and propositional resources available at the initial state are decreased by removing available predicates or restricting the valuation of numeric resources.

We describe each domain model and their associated set of problem instances as follows:

4.6.1.1 Cashpoint

We introduce two variants of the original cashpoint domain used in Chapter 3: *simple cashpoint*, and *cashpoint rewards*. The purpose of using these domains is to explore the behaviour of planners where there is a wide range of action choices available while achieving various different goal conditions. The simple cashpoint domain is almost identical to the original cashpoint domain. The objective is to acquire a certain amount of money in different currencies from the bank (or exchange bureau) and to supply various items (e.g. buying snacks) using the money withdrawn. There are two options to reach the monetary goal: either by withdrawing it at any of the banks (if the bank has the currency in their reserves) or by swapping the currency in the pocket at any of the exchange bureaus (using fixed exchange rates between currencies). In cashpoint rewards domain, the following features are included on top of the simple version. First, the person can exchange currencies between their accounts in any of the banks to the extent of their available balance. Second, the person collects reward points every time they shop and can use it as store credit on their next purchase. Third, the person can pay using their debit card at the shop so that they do not need to use cash in the shop. These additions give further freedom of action choices

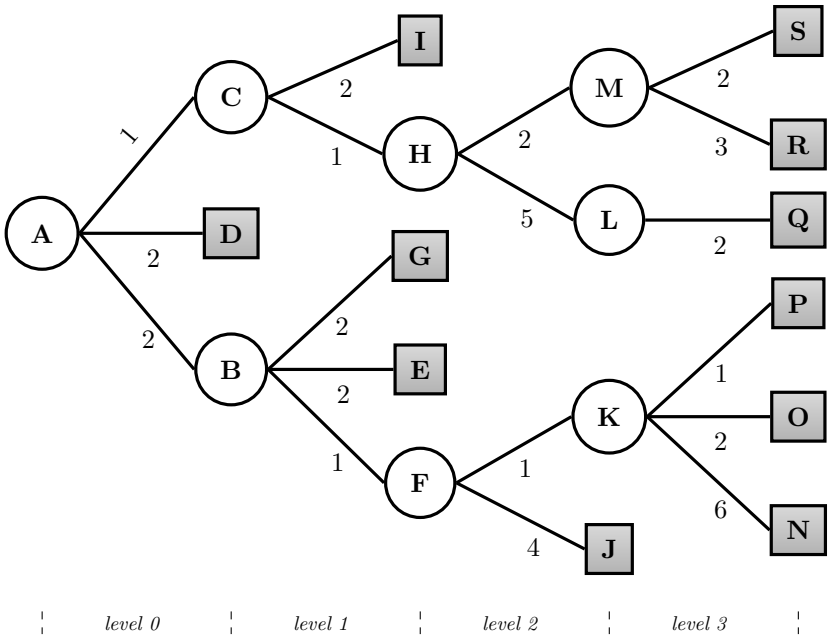


Figure 4.12: The complete bill of material tree of the procurement domain model.

to the planner so that the search space is expected to be wider. In both domains, as the amount of cash required on each currency increased relative to the cash point limits, the plan has to extend to include additional withdrawal or exchange actions.

This domain is an ideal instance of a planning problem with a single control parameter per action. Although there is producer-consumer relationship between actions (exchange between bank accounts and money withdrawal actions), the primary focus of using this domain is to identify how our approach behaves in a domain with rich typed objects. In both domains there are five types of objects defined: *currency*, *shop*, *bank*, *change office* and *item*. Since each action is instantiated with a combination of its objects, this feature broadens the width of the search tree, and results in relatively proliferated branching choice. It is also worth noting that the duration of each action is constant.

4.6.1.2 Procurement

The procurement domain is a complex version of our motivating example. It is about finding a plan to manufacture final products, which require different amount and types of raw materials to be procured from a store and intermediate products to be produced at a workshop.

The purpose of using this domain in our tests is to examine the great extent of cascading producer-consumer relationship between actions using a single control parameter per action. The control parameter manages the amount by which an item can be supplied or produced per execution of its action. The upper bound of each control parameter is restricted by linear functions of consumed state variables resulting in a set of linear constraints over the control parameter. Unlike cashpoint domains this domain also contains a deeper search space, producing longer plans, allowing us to explore the cost of solving increasing numbers of more complex constraint sets.

Figure 4.12 shows the complete material requirement tree, also known as *bill of material* tree, of this domain. This tree is an extended version of the original procurement domain (presented in Chapter 3) that is higher in depth and consists of additional nodes. There are in total 8 final/intermediate items to be produced and 12 raw materials to be supplied. The square nodes represent raw materials, which can be obtained from the store. The circular nodes represent the intermediate and final products. To produce a product A, we need to have already sub-assembled two product B, a product C and have already acquired two raw material D. Product B and C are the intermediate products that require products F, E, G, H and I. In this domain, the batch sizes of items procured from the store and produced at the workshop are taken as control parameters where each can take any value up to one hundred units.

Each problem instance is generated with a random problem instance generator. The complexity of each instance depends on four inputs: the number of workshops, customers, suppliers and the number of product goals to be delivered to each customer. The generator chooses each product to be delivered (to a customer) and each customer randomly from the items and customers list, respectively. It also assigns the initial numeric value and the numeric goal of the product with an integer value up to 15 and 50 units, respectively. The lower the level (as seen in Figure 4.12) and higher the numeric goal of the product, the harder the problem instance gets. As the material hierarchy given in the figure is fixed in the domain model, the item list does not vary between problem instances.

4.6.1.3 Terraria-Capacity

We previously introduced this domain in Chapter 3. The objective in this domain is to find a plan for producing certain products (as in the procurement domain). There are three features of this domain that differs from the procurement domain. First, it includes multi-variate capacity constraints that limit the total amount of raw material procured. Second, the plan may include assembly of some machines or tools to be used for producing items. For instance, sawmill needs to be available in order to produce a bed. If it is not available, the plan needs to include a sequence of actions for assembling a sawmill. This feature creates a mixed propositional (i.e. having sawmill ready) and numeric causal link between actions where producer-consumer relationship still exists. This allows us to examine the completeness of the refined infinity analysis when partial ordering in relaxed planning graph is necessitated (whether the producer-consumer patterns can be partially ordered). Third, there are multiple control parameters in individual actions in this domain. The complexity of each Terraria problem instance is extended by decreasing the capacity limit of raw materials, and similar to procurement domain, by increasing the amount of required final/intermediate products from its material/equipment requirement tree.

Although the domain syntactically fits to show producer-consumer behaviour, the original problem instances (in Chapter 3) were not challenging enough to trigger producer-consumer behaviour (the replenishment of consumers were not needed). Thus, we added eight new numerically challenging problem instances to the original problem instance set so that we can test the domain when the producer-consumer behaviour is triggered. The complexity of problem instances is increased by adding more numeric conditions to be achieved and increasing the required numeric limits of these goal conditions.

4.6.1.4 Rovers Control

This domain is a modified version of the well-known Rovers domain introduced in IPC-3 (Long and Fox, 2003). In this domain we add the following features. First, the recharge action is modified so that the numeric transfer occurs with a control parameter, *?recharge_amount*, which can take any value up to the available energy capacity of the rover (at most eighty). Second, we add an action replenishing the energy level at the recharge station: *turn_on_solar_panels*. If there is insufficient energy at the station, the solar panel needs to be turned on so that the recharge station can produce energy. This addition creates an interesting producer-consumer relationship between two actions where they are both constrained with capacity limits (energy capacity of the recharge station and the rover). Third, the lower bound of the duration in both recharge and running solar panel actions is constrained with a control parameter. This linear relationship reflects the fact that the duration of each durative action depends on the lowest recharge amount chosen by the planner.

4.6.1.5 Airplane Cargo

This domain is based on the well-known logistics domain used during The First International Planning Competition (IPC-1) (McDermott, 2000). In this domain the goal is to find a plan to deliver various cargo packages to destinations via air planes. We made the following modifications to the benchmark domain to examine the case with control parameters. First, all actions are temporal and can require concurrency of actions during execution. The duration of each action is either fixed or a function of the distance between start and end locations. All actions are typed (e.g. *tanker, airport, refinery, cargo, plane*) that enables concurrency between different instantiations of an action. Second, we added fuel tankers that refill the fuel tank of air planes with a control parameter to make sure it has enough fuel to arrive at a destination. To do this, the tanker first may need to refill its tank by the required amount at a refinery station. This modification creates an interesting producer-consumer relationship between actions, which is subject to tank capacities of the tankers and air planes.

In this domain all numeric changes of actions are discrete. We define two fly actions (i.e. *fly_airplane* and *fly_airplane_fast*) where the travel time and fuel change rate varies. In case fast option is chosen the travel time reduces while the fuel consumption increases. This option creates a further plateau of action choices in the search space. The planner is expected to choose an option which favours the metric objective function defined in each problem (in our case it is to minimise total time). The fuel level of each air plane is decreased by a linear function of the distance between locations in fly air plane actions.

We included this domain in our tests for the following qualities. All goals defined in problem instances are propositional so that we can survey the behaviour of refined infinity analysis when satisfying propositional goals. The domain is relatively rich in terms of number of types defined and typed objects of actions. The required concurrency feature of the domain provoke finding interesting plans.

We randomise the following artifacts when generating each problem instance:

– The initial locations of fuel trucks, air planes, cargo packages,

– The fuel levels of fuel trucks, air planes and fuel levels of each refinery locations,

– Driving and flying distances between locations.

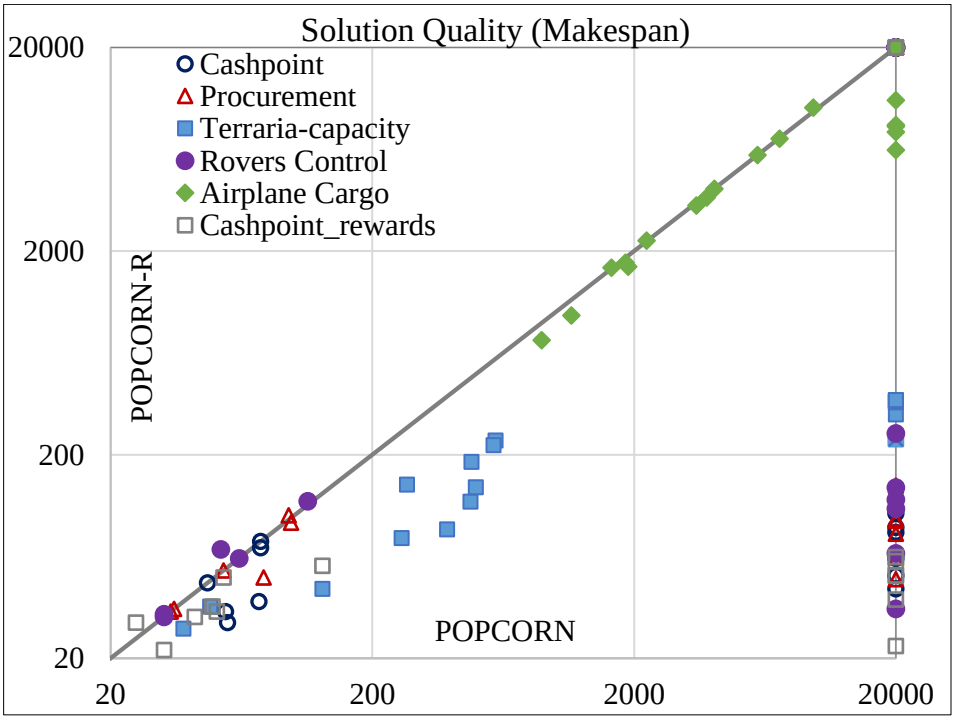


Figure 4.13: The solution quality (makespan) results of POPCORN and POPCORN-R[7].

4.6.2 Performing Ablation Study

Having described the domain models and sets of problem instances, we perform ablation studies using 6 different domain models, where each having 20 random problem instances. Our objective in this study is to determine whether our approach:

– improves the running performance of the system (*scalability*),

– can solve problems that have been previously identified as unsolvable (*solvability*),

– can fit to other components of the base system that are critical in planning perspective, such as required concurrency and mixed propositional-numeric goal reasoning (*compatibility*).

Figure 4.13 compares the solution quality of the new planner POPCORN-R and the POPCORN in logarithmic scale. Observe that POPCORN-R provides shorter plans to almost all solvable problems when compared to POPCORN. This improvement states that our approach avoids considering arbitrary executions of actions as it is well-guided about the occurrence limits in the search space. The most noticeable improvement in solution quality is

[7]The points on the edges of the graph indicates that only one of the planners were able to find solution to the corresponding problem instance. The points on the right top corner were unsolvable to both planners.

observed in terraria-capacity domain, which requires intensive mixed propositional-numeric goal reasoning. In other words, this domain is rich in both numeric and propositional sub-goals, where the planners are expected to reason with both sub-goaling perspectives. This complication is apparently mishandled by POPCORN, where it *blindly* adds actions to the plan to be able to find *a* solution.

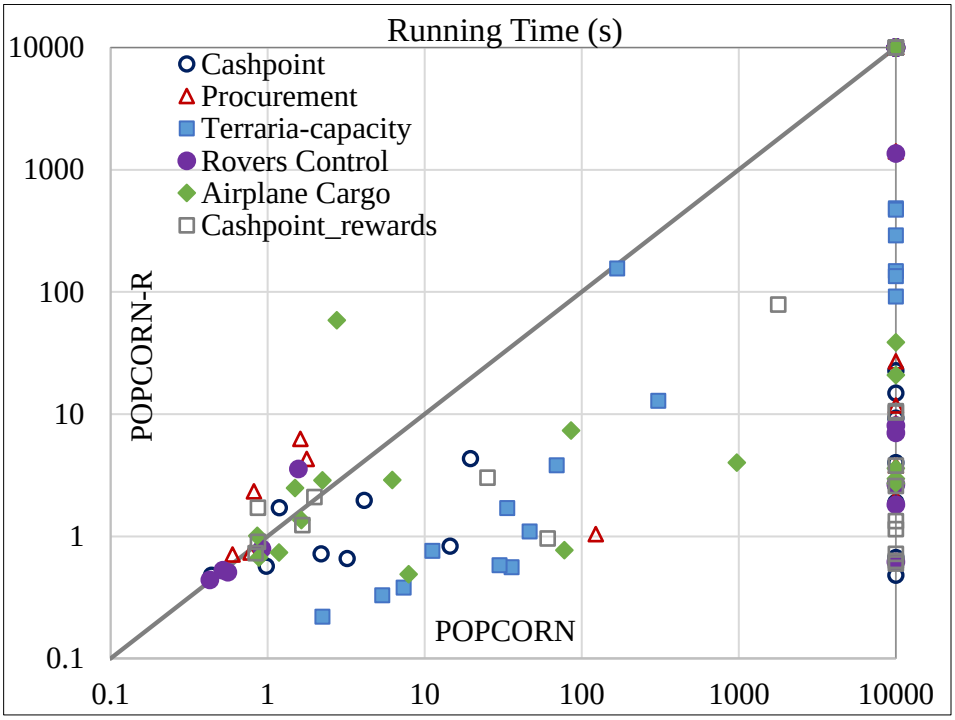


Figure 4.14: Running time results of POPCORN and POPCORN-R.

Running time comparison between systems is given in Figure 4.14. Observe that POPCORN-R is faster and scalable than POPCORN in more than half of the solvable problem instances. However, some of the solvable instances support that POPCORN-R is relatively slower than POPCORN. We can comment on this result based on three aspects: the use of LP in the heuristic, the overhead cost of refined infinity analysis and poorly guided (and thus exhaustive) search of POPCORN:

The use of LP: The modified TRPG heuristic of POPCORN (described in Section 3.5.6) makes use of solving LP models to compute the bounds of non-constant bounded control parameters once in the heuristic. We eliminated this use in the refined infinity analysis. By doing this, we presume that the use of LP in POPCORN-R is 2 or 3 times less than the one in POPCORN, where each LP call carries a cost.

Observe that POPCORN is slower in almost all problem instances of the terraria-capacity domain than POPCORN-R. Recall that this domain includes actions with multiple control parameters (that are non-constant bounded). It is self-evident that in this domain POPCORN spends considerable amount of time solving LP models, while POPCORN-R does not.

Overhead cost of refined infinity analysis: The overhead cost of our approach becomes apparent on test cases that both systems get equally-informed guidance. These

cases are the ones that are cluttered around the baseline (or slightly skewed to the up left). This indicates that refined infinity analysis slightly increases the mean scheduling time of the heuristic (time spent in heuristic). This is an expected result as our approach downgrades some of the major relaxations imposed in the heuristic (i.e. arbitrary occurrence limits).

Suffering from exhaustive search: The results indicate that POPCORN is noticeably slower than POPCORN-R in most problem instances. This situation results from the fact that POPCORN performs exhaustive search on these problem instances, because its heuristic either promotes wrong activities as helpful or fails to estimate true occurrences of actions. This problem becomes more recognisable in problem instances that create wider search spaces. Tests on cashpoint and cashpoint_rewards domain models can be shown as an evidence to this case as they create wider search space than other domain models. Observe that POPCORN tends to get slower in numerically harder problem instances, because the planner does not get adequate guidance at a state where there is a wide range of action choices. Although terraria-capacity domain does not create as wide search space as cashpoint domains, the complex goal reasoning of the domain causes an exhaustive search to almost all solvable problem instances. One possible explanation to this behaviour is that POPCORN fails to recognise propositional sub-goals that should be triggered by numeric goals (or sub-goals).

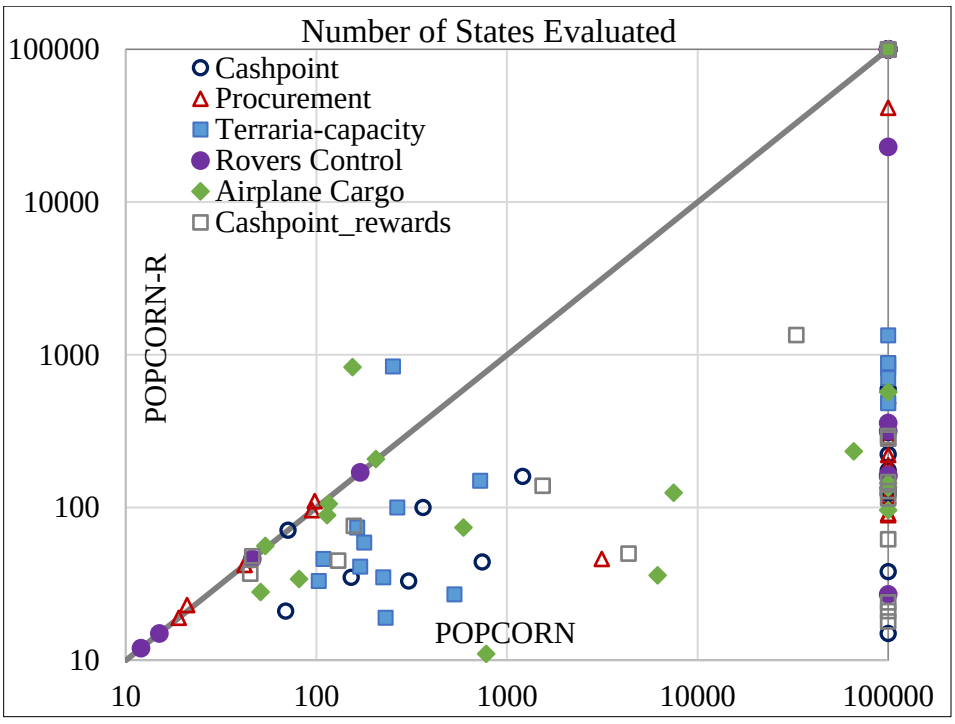


Figure 4.15: Number of states evaluated of the planner in six domains (POPCORN vs. POPCORN-R).

Although our approach holds a minor overhead cost, it yields superior results in the end. The comparison of the number of states evaluated is an ideal indicator of the improvement. It reveals the number of times the systems compute heuristic for solving each problem instance. Figure 4.15 exhibits this comparison between POPCORN and POPCORN-R. Observe that

POPCORN-R plummets the number of heuristic computations in almost all solvable problem instances. This result supports that POPCORN-R makes intelligent decision on choosing states to progress. We noticed that POPCORN makes highly random state progression decisions. We can detect the randomness from the figure: the results of POPCORN show a significant variation for most domain models. For instance, it varies between 70 states to almost 90000 states in airplane cargo domain, while POPCORN-R finds a solution to almost all its solvable instances (90 out of 94) by evaluating less than 1000 states.

Table 4.2: Number of problem instances solved by POPCORN and POPCORN-R.

Domains:	cashpoint	procurement	airplane	rover	terraria	cashpoint_rewards
POPCORN	8/20	6/20	12/20	5/20	11/20	8/20
POPCORN-R	16/20	15/20	17/20	11/20	18/20	17/20

Some of the problem instances are deemed unsolvable to the systems in our experiments. Table 4.2 presents the number of problem instances solved by each system. We used 20 problem instances per domain model in our tests. The last 5 instances on each domain model are designed to be extremely hard problems. The most challenging domain in our tests is rovers domain as it includes highly complex causal and numeric goals. We kept the difficulty of goals the same as the original benchmark version and extended it to include control parameters. POPCORN and POPCORN-R can solve 50 and 94 problem instances, respectively, out of 120 instances in total. Notice that the solvability of POPCORN has almost doubled with refined infinity analysis. As numeric goals get harder, the POPCORN fails to handle complex numeric and propositional sub-goaling and is led to dead-ends. Overall, we can say that POPCORN-R can handle hard (and some of extremely hard) problem instances whereas POPCORN can handle easy-medium problems and fails on hard instances.

4.6.2.1 Ablation Study on Unmodified Domains

The ablation study benchmark domains are the modified versions used in Chapter 3. We tested our approach on the unmodified versions (exactly the same domain and problem sets of Chapter 3) between POPCORN and POPCORN-R to check whether our work restricts the solvability of the base system. Recall that unmodified versions are considerably easier than the modified versions in general. We obtained identical results on logistics and rovers domain and noticeable improvements on cashpoint, procurement and terraria. Figure 4.16 illustrates the comparison between planners in these domains described in Section 3.6. This outcome supports the fact that POPCORN uses an LP solver in the heuristic, whereas POPCORN-R does not.

4.6.3 Comparison with Scotty Planner

The Scotty and POPCORN planners are the only systems available that can reason about PDDL domains with control parameters. We made various modifications on the motivating example used in Scotty, *the AUV domain*, so that it generates producer-consumer situation between actions. We call the resulting domain *AUV-Fuel*. We split the *glide* action to

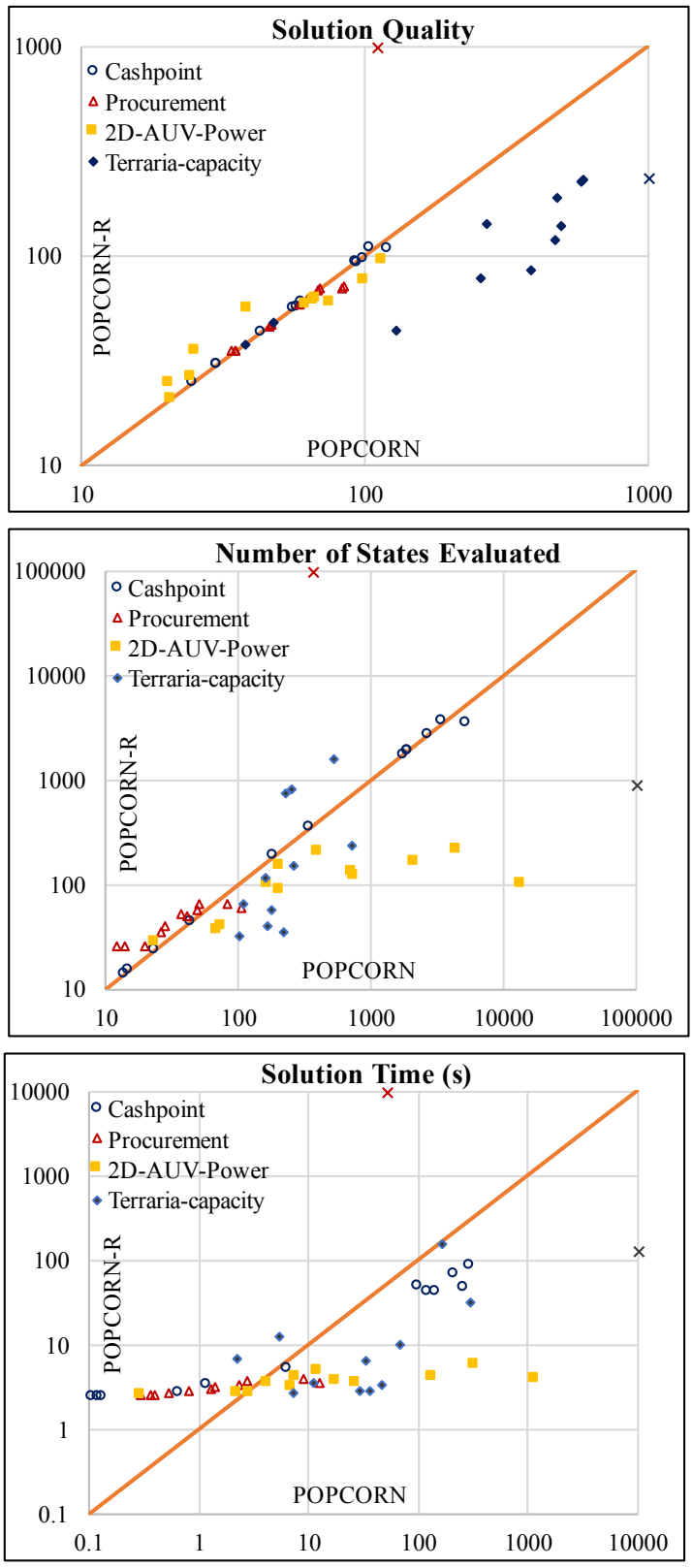


Figure 4.16: Ablation study results on unmodified versions of the domains (the same domains used in Section 3.6). The "x" markers indicate that only one of the planners could solve the instance.

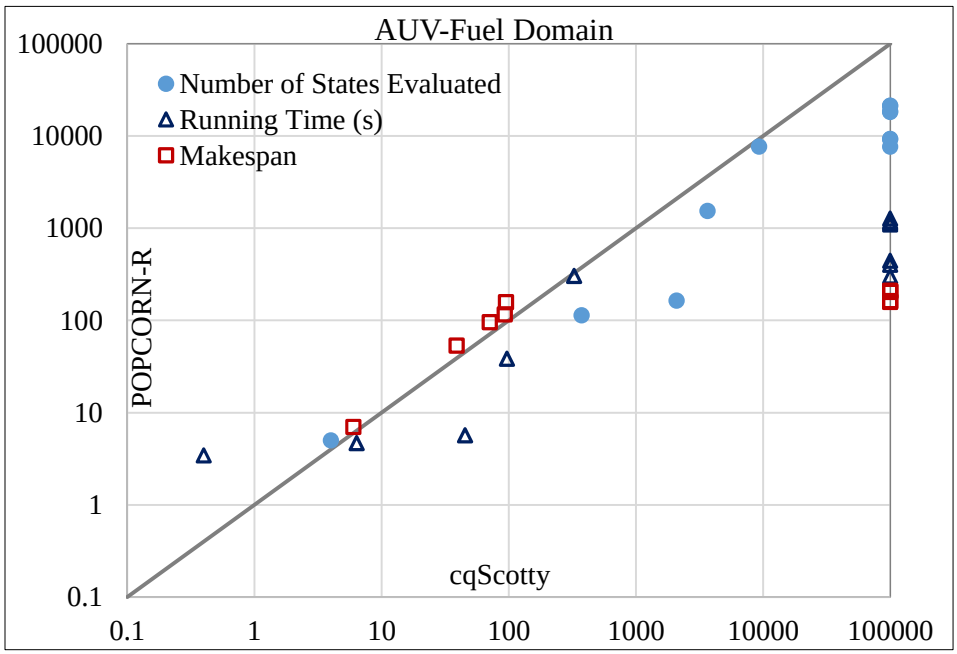


Figure 4.17: The makespan, the running time and the number of states evaluated results for the AUV-Fuel domain of POPCORN-R and cqScotty.

glide_backward and *glide_forward* in order to avoid ambiguous plan generation due to existence of negative control parameters available in the original domain. We added various resource replenishment actions (i.e. *recharge*, *turn_solar_panels_on*) so that it arises basic producer-consumer relationship. We compared the POPCORN-R and cqScotty on this domain with 12 problem instances. In cqScotty, the resource transfer occurs with linear continuous numeric change where the control parameters are the rates of change of the action duration. In POPCORN-R, the numeric change is discrete and control parameters are duration-independent. The results are shown in Figure 4.17. The POPCORN-R solves all while cqScotty solves 5 of the problem instances. We observed that the EHC fails and the search resorts to BFS for all problems in both systems, which leads to an exhaustive search and an increase in state evaluation. This situation results from the fact that both approaches employ RPG-based heuristics. Their heuristics fail to promote *glide_backward* action because of the delete-relaxation assumption.

4.7 Summary

In this chapter, we presented a capable approach for providing guidance to forward chaining $P-$ and $C-$state space search. The approach is a generalised guidance solution for handling expressive metric-temporal planning domains with control parameters, which is relevant to many real world application fields. The refined infinity analysis approach we propose is an alternative sub-goaling method for planning domains that have no control parameters. However, it almost is a necessity for domains with infinite numerical action parameters. Alternative solutions to well-known numeric planning problems (i.e. the producer-consumer behaviour) are shown to be either highly expensive (i.e. LPRPG) or less-informative than

our proposed solution. Our proposed approach can be easily employed to any planner that uses a TRPG-based heuristic (including Scotty) to overcome helpful action distortion. While the producer-consumer patterns we introduce cover a broad class of problems, there are other expressive patterns to explore in the future.

Chapter 5

Case Study: Spatial Reasoning in Task Planning with Control Parameters

5.1 Introduction

In recent years, the research communities in planning and robotics are highly involved in the subject of the integration of task and motion planning (TAMP). In general this integration works as follows. A task planner constructs a semantically meaningful (yet numerically unparameterised) symbolic plan while the motion planner finds a collision-free trajectory. Figure 5.1 illustrates task-motion planning interaction in robot navigation. Although there is a rising number of contributions dedicated to this topic over the last decade, bridging the gap between the two remains unsolved. In particular, while the task planner reduces the continuous dynamics of the environment into discrete parameters, the motion planner reasons with its full complexity. The task planner loses critical knowledge about the dynamics during the reduction, which usually results in significant sacrifice on the intelligent behaviour. The rationale behind the discretisation is to avoid infinite branching

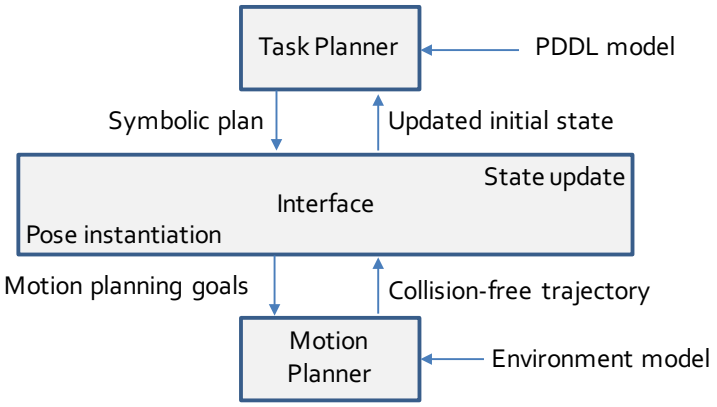


Figure 5.1: Schematic representation of the integration of task and motion planning.

```
(:durative-action navigate
 :parameters(?r1 - robot ?from ?to - waypoint)
 ...)
```

Figure 5.2: The navigate action with discrete location parameters.

choice of actions in search, however this can also be slow and inefficient when the number of grounded actions is relatively large.

The planning paradigm we propose in this thesis extends the capabilities of task planners in the TAMP integration. It enables the modeller to model locations as *continuous regions* by defining a set of numeric constraints over the coordinates of robots and objects (taken as state variables) and predefined flexible limits of these coordinates (using control parameters). The POPCORN planner accumulates these constraints in the constraint space. In this integration, POPCORN generates a controlled plan π (i.e. timestamped actions with flexible control parameters) subject to temporal-numeric constraints accumulated during planning. The motion planner assigns a value to each flexible control parameter iteratively using the controlled plan management (presented in Section 3.5.5.1) *during plan execution*. If the motion planner cannot satisfy π with the kinematic constraints, then the initial state needs to be updated with new bound limits of these constraints and the planner needs to find a new plan for the updated initial state. This mechanism conveniently fits to the state update layer illustrated in Figure 5.1.

As a concrete example, showing that discretisation is not always computationally appropriate, consider the action of navigating a robot from a waypoint to another (see Figure 5.2 for the action schema) and in a problem description there are 10 robot and 40 waypoint objects. This action is grounded by the substitution of parameters for each possible value defined in the problem description. This consequently leads to a situation, at which the planner has finitely many action choices in the search, yielding to at most $10 \times 40 = 400$ grounded navigate actions at each depth. Figure 5.3 illustrates the situation in the forwards search. This situation becomes a combinatorial search problem and the exhaustive search can be impractical to solve it (yet, exhaustive search is inevitable if all possible states are deemed as equally helpful by the heuristic at a state).

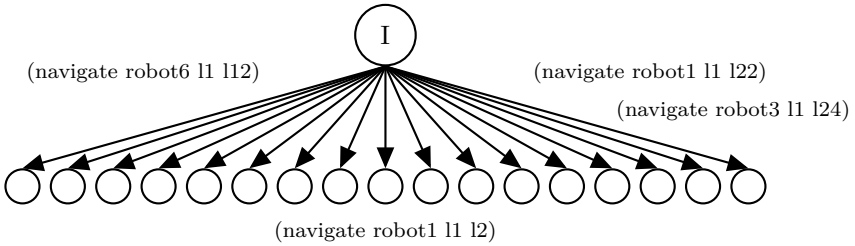


Figure 5.3: Illustration of navigate action choices with discrete locations in forwards search. Each edge in the search graph corresponds to a grounded `navigate` action (given in Figure 5.2) with respect to 10 predefined robot (e.g. $\{robot1, \ldots, robot10\}$) and 40 location (e.g. $\{l1, \ldots, l40\}$) objects.

In addition to the computational complexity, exploiting discrete locations is brittle to the physical dynamics of the environment. It often leads to plan invalidation, re-planning

```
(:durative-action navigate
 :parameters(?r1 - robot)
 :control(?from ?to - number)
 ...)
```

Figure 5.4: The navigate action with continuous location parameters.

during execution or another consequence would be observing unintelligent behaviour due to poor quality plan. For instance, in case the discrete waypoint to be visited is physically blocked so that the robot cannot visit it, the robot will have to re-plan. The rationale behind these problems is that the task planner ignores the geometric constraints imposed on activities such as in robot navigation because it is handled by the motion planner. In order to fill this gap between task and motion planning, in this chapter, we propose replacing the discrete location representation with *c*ontinuous regions modelled using control parameters. To accomplish this we model the location parameters (i.e. `?from`, `?to` in Figure 5.2) as linear functions of control parameters, so that:

1. The combinatorial size of grounded actions can be significantly reduced in search.
2. The task planner would have a realistic estimate of the configuration space dynamics.

Figure 5.4 shows the navigation action schema of the proposed approach. In the example we presented earlier the search space size decreases from 16000 fully-grounded states to 10 partially-grounded states. Figure 5.5 illustrates the resulting search space.

The chapter is structured as follows. We present the observe-and-classify scenario considered in SQUIRREL EU Project as our motivating example in Section 5.2. We propose a modelling technique to define continuous regions using control parameters in Section 5.2.1. We showcase how reasoning with continuous regions is scalable (even if it relies on LP support) and significantly improves the solution quality with detailed experimental results in Section 5.4.

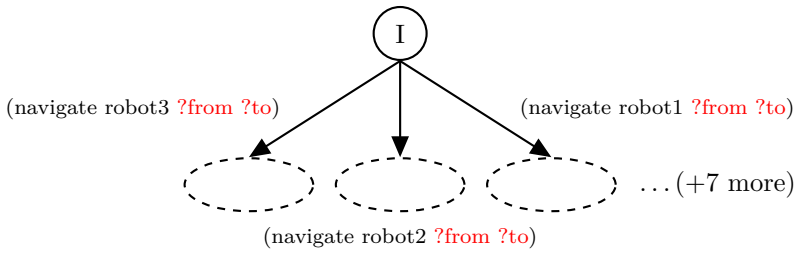


Figure 5.5: Illustration of partially-grounded navigate action choices with continuous locations in forwards search.

5.2 Observe-and-classify Scenario

We now present a scenario used in the SQUIRREL EU robotics project. Consider that a robot travels around in a two dimensional bedroom-like environment to classify toy objects scattered around the room. The robot initially needs to confirm the coordinates of each

```
(:action move
 :parameters(?from ?to - location)
 :precondition (and (connected ?from ?to) (robot_at ?from))
 :effect (and (not (robot_at ?from)) (robot_at ?to)))

(:action observe
 :parameters(?loc - location ?obj - object)
 :precondition (and (robot_at ?loc) (can_observe ?obj ?loc))
 :effect (and (observed ?obj)))

(:action classify
 :parameters(?loc - location ?obj - object)
 :precondition (and (observed ?obj) (robot_at ?loc)
    (can_classify ?obj ?loc))
 :effect (and (classified ?obj)))
```

Figure 5.6: The STRIPS domain model of the observe-and-classify scenario.

object using visual sensors before it can get closer to grasp and classify the object. Thus, the robot first gets closer enough to the coordinates where the object is initially assumed to be located so that it can perform the *observe* action. Once it is done, the robot gets even closer to the object to perform the *classify* action.

The scenario can be modelled in STRIPS as follows. The waypoints at which the robot can execute `observe` and `classify` actions can be defined as discrete action parameters (i.e. object parameters) in the PDDL model, and each possible value these parameters can take must be enumerated as an object in the problem description. The STRIPS domain model is given in Figure 5.6. In this domain, the robot moves from one waypoint to another using the `move` action. Most off-the-shelf classical PDDL planners, including (Hoffmann and Nebel, 2001; Bonet and Geffner, 2001; Helmert, 2006), can reason with this model quite efficiently in a problem with a few tens of objects.

The scenario can also be modelled as a numeric planning model in PDDL2.1 (level 2). The coordinates of the each waypoint (of the objects and the robot) can be defined as discrete numeric variables, which are explicitly enumerated in the problem description. Figure 5.7 shows the actions of the model in detail. The values of the state variables used in this model are initially enumerated in the problem and they are always known at any time point in search. This contrasts to models with control parameters, in which they are kept flexible. For this reason, the robot must be precisely at the same coordinates with the object to be able to execute the corresponding action (using numeric equality conditions). As discussed in Section 2.2.2, numeric planning yields to more complex branching choice than the one in classical planning. This observation makes clear that this model is computationally harder than the STRIPS model.

Both of the models presented express waypoints as discrete notions that are only allowed to take their values from a domain explicitly defined by the modeller. Figure 5.8 demonstrates an example layout for these modelling scenarios (i.e. classical and numeric). The robot can observe the toy car at one of the black dotted waypoints (either defined as objects or coordinates) while it can classify at one of the red dotted waypoints.

```
(:action move
 :parameters(?from ?to - location)
 :precondition (and (connected ?from ?to)
   (= (x_robot) (x_location ?from)) (= (y_robot) (y_location ?from)))
 :effect (and (assign (x_robot) (x_location ?to))
   (assign (y_robot) (y_location ?to))))

(:action observe
 :parameters(?loc - location ?obj - object)
 :precondition (and (can_observe ?obj ?loc)
   (= (x_robot) (x_location ?loc)) (= (y_robot) (y_location ?loc)))
 :effect (and (observed ?obj)))

(:action classify
 :parameters(?loc - location ?obj - object)
 :precondition (and (observed ?obj) (can_classify ?obj ?loc)
   (= (x_robot) (x_location ?loc)) (= (y_robot) (y_location ?loc)))
 :effect (and (classified ?obj)))
```

Figure 5.7: The numeric model of the scenario encoded in PDDL2.1 level 2.

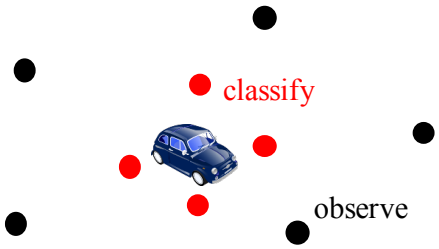


Figure 5.8: Schematic representation of the discrete locations defined around a toy.

5.2.1 Modelling Continuous Regions

Having presented discrete modelling solutions for the off-the-shelf planners, we now present a continuous solution that allows a task planner (integrated in a TAMP system) to reason about the geometry of the environment during planning. As previously mentioned, instead of modelling locations as *discrete notions* using variables (or action parameters) with finite sets, we model them as *continuous regions* using control parameters. Suppose that we define a circular area around the object as a *collision-free* space and it is surrounded by two ring-shaped areas where the robot can execute `classify` and `observe` actions in each independently (all areas are concentric). Figure 5.9 illustrates the description. Conceptually, landing on any point within a region is adequate to apply the corresponding action during execution.

The regions are modelled as numeric conditions of actions in PDDL model as follows. The move action has two real-valued control parameters (i.e. $\{?dx, ?dy\} :: [-100, 100]$), which represent the displacements of the robot in x- and y- directions. The move action increases/decreases the coordinates of the robot with on each direction by linear functions of $?dx$ and $?dy$ (i.e. `(at end (increase (x_robot) ?dx))`). Thus, applying one move action is sufficient reaching to any coordinates in a 100×100 sized environment. Also, the duration

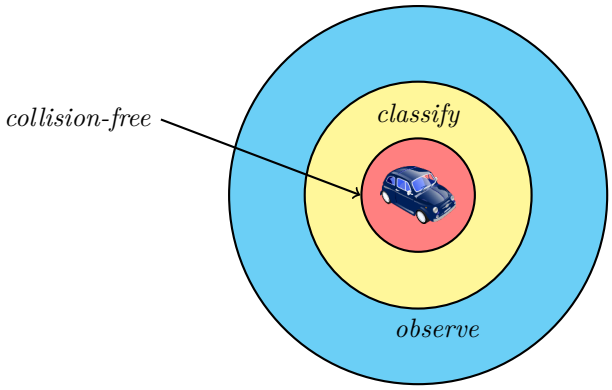


Figure 5.9: Continuous locations around the toy.

of the action is flexible and it is constrained by a linear function of ?dx and dy (which yields to the Manhattan distance).

The observe and classify actions have sets of numeric preconditions each defining convex regions of the corresponding location. The numeric preconditions form circumscribed octagons as they are linear over-approximations of circular regions. The reason behind using octagons is that having circular regions create non-linearity which is currently not handled in POPCORN. Suppose that a circle centered at (x_w, y_w) coordinates and it has a radius of r. The Euclidean distance (d) of any point, (x_r, y_r), within any circle to the center is $d = \sqrt{(x_w - x_r)^2 + (y_w - y_r)^2}$. Equation 5.1 consists of a set of numeric constraints that approximates the distance (using octagons). In our scenario, (x_r, y_r) and (x_w, y_w) denotes the coordinates of the robot and the initially predicted coordinates of the object, respectively. The full PDDL domain model with continuous regions that is used in our experiments is given in Appendix B.2.1.

$$\begin{aligned} -r \leq |x_w - x_r| \leq r \\ -r \leq |y_w - y_r| \leq r \\ -\sqrt{2}.r \leq |x_w - x_r| - |y_w - y_r| \leq \sqrt{2}.r \\ -\sqrt{2}.r \leq |x_w - x_r| + |y_w - y_r| \leq \sqrt{2}.r \end{aligned} \tag{5.1}$$

Observe that, in Equation 5.1, some expressions are given in the form of absolute values. The PDDL extension we use in this work does not allow encoding absolute values, thus the equation set must be split into four possible sets with regard to the signs of variables. Then, we define 4 observe and 4 classify action schemas in the domain encapsulating each possible set. For instance, for classify action we model *classify_southeast, classify_southwest, classify_northeast, classify_northwest* action schemas, where each of these action schemas cover a quarter area of each polygon. Figure 5.10 demonstrates 12 quarter-polygons each covered by individual actions.

In addition to convex region constraints, the observe and classify actions have a literal condition (i.e. `(empty_northeast ?object)`) controlling whether the linked continuous quarter-polygon region is occupied by an obstacle. If so, it is set to be false at the initial

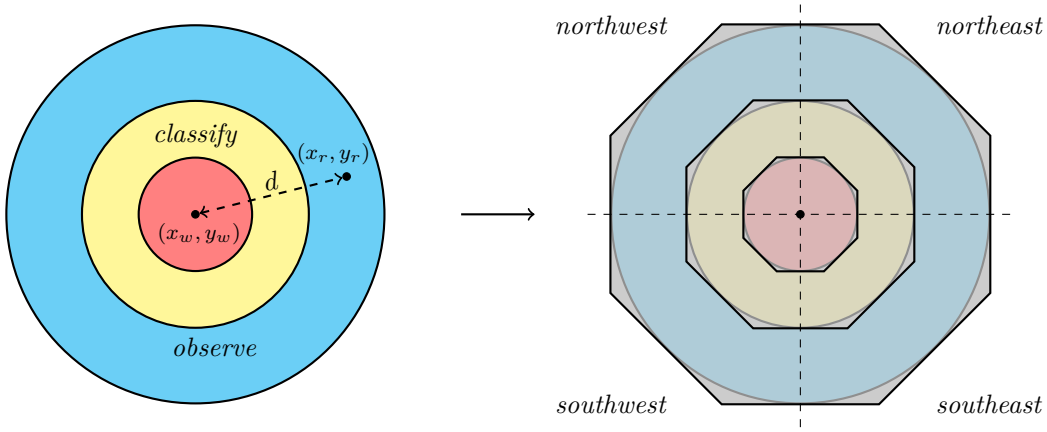


Figure 5.10: Illustration of linear over-approximation of regions using circumscribed octagons.

state, so that the robot does not visit the region.

There are certainly other ways to model the continuous regions in PDDL. The model we showcase here is an ideal instance for handling the scenario we presented. It is also worth mentioning that the use of control parameters is not only limited to defining continuous regions in robotic applications. There are various other dynamics that require flexibility and they can be suitably modelled with control parameters. For instance, suppose that another scenario requires PDDL modelling to control the movement speed of a robotic arm. If the speed parameter is not modelled as flexible and the modeller defines only a few numeric choices for it, the motion planner will immediately invalidate the decision drawn by the task planner. The modeller must be able to foresee the precise value for it, so that the generated plan remains valid during execution.

5.3 Plan Generation and Execution

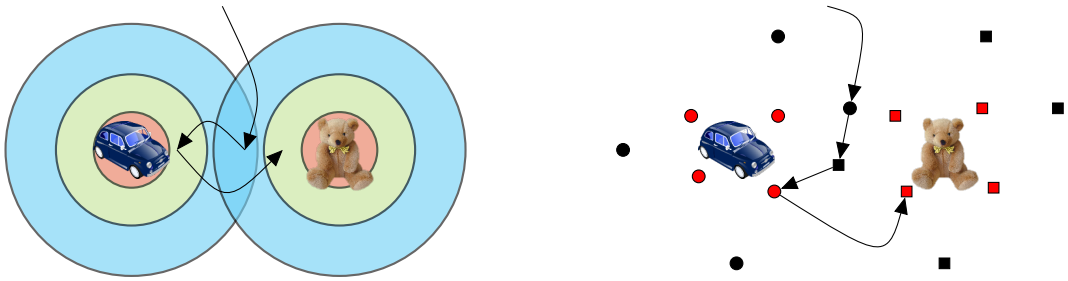


Figure 5.11: The illustration of the plan generation when the objects are in close range.

5.3.1 Plan Generation in Relative Proximity of Objects

One of the many benefits of reasoning with continuous locations is that it provides flexibility to the planner when satisfying action conditions. To make this statement clear, consider the following situation in robot navigation. Some areas in the configuration space can

be common to more than one continuous region. Any point within this intersection is a member of all the overlapping regions, so that all actions conditioned on them become applicable. This flexibility can enable eliminating excessive navigation actions in the plan, which can consequently lead to a noticeable improvement in the solution quality. As the planners attempt to keep the plan length minimal by default, they favour navigating to these intersections. From the planning perspective, choosing intersections is a constraint satisfiability (and optimisation) problem rather than a combinatorial search issue.

We can exemplify this concept in our running scenario. Consider that two objects are relatively close to each other so that their observe regions are not disjoint. The scenario is illustrated in Figure 5.11 for both continuous and discrete cases. Figure on the left shows the intersection, at which the robot can observe both toys. The goal of classifying both objects can be reached in 7 plan steps: *move, observe car, observe bear, move, classify car, move, classify bear*.

The figure on the right shows the same scenario in the discrete case. In this figure, the circle and the rectangle dots denote waypoints for the car and the teddy bear toys, respectively. The red and black waypoints are the ones that the robot can apply classify and observe actions, respectively. The same goal can be achieved in at least 8 plan steps in the discrete environment, which includes an additional move action than the one generated in continuous environment.

Consider a situation when the objects are even closer so that their classify regions can overlap with each other. While the solution obtained in the discrete environment remains unchanged, it is achieved in 6 plan steps (one less move action) in the continuous environment. We analyse the effects of the relative distance on the solution quality in Section 5.4.2.

5.3.2 Plan Execution using Controlled Plan Management in Continuous Space

We presented the advantages of transforming a robotics environment into a continuous formalism from a task planning point of view. In this section, we survey a possible execution mechanism of the plans with continuous locations.

In the POPCORN planner, the continuous regions are formed in the constraint space. The generated plan provides the maximum and minimum values of the control parameters computed from the final collection of constraints accumulated during planning. As the bounds can be inter-dependent, we proposed an iterative search mechanism, controlled plan management, that updates the bounds for each numeric valuation decision made by the executive. This process is presented in Section 3.5.5.1. When exploiting the controlled plan management in this robotics application scenario, the motion planner can act as the executive, where it makes coordinate decisions within the supplied bounds iteratively *during execution*. The rationale behind applying this search during execution is the following:

1. There are uncertainties that can alter the trajectory plan to a great extent: the initially claimed feasible regions can be discovered to be unreachable during execution. This can even result in invalidation of the symbolic plan, yet this remains the greatest question unresolved in TAMP applications.

2. Since the motion planner does not possess the full knowledge of the feasible configuration space, making valuation decisions before the execution is likely to result in losing plan flexibility. This increases the possibility of re-planning during execution.

The only difference between making decisions during and before execution is the following. If the motion planner needs to backtrack in the controlled plan management (i.e. revisit its previous coordinate decisions), the system might need to generate a new controlled plan as it might be impossible to revert to previous model states. In other words, the backtracking can result in navigating back to a previously visited waypoint to execute the revised decision.

The location coordinates in this scenario is two-dimensional, so each coordinate decision enforced by the motion planner includes two numeric valuations (that may be inter-dependent) at the same time. In this case, the controlled plan management can perhaps break apart this combined decision into 2 individual decisions and return the final updated bounds to the motion planner.

5.4 Evaluation

To showcase our approach, we conducted a detailed set of task planning experiments comparing the effects of encoding locations as discrete notions and convex regions based on observe-and-classify scenario. We use POPF and TFD planners to solve discrete problems on a propositional-temporal PDDL model and the POPCORN planner to solve problems with continuous regions. The PDDL domain models used in the experiments and example problem instances are given in the Appendix B.1 and the Appendix B.2.

5.4.1 The Environment

In order to make a fair comparison between different approaches, we define problem instances with different sized regions and varying granularity of discretisations. Overall, our study involves 1600 problem instances with different features and they are listed as follows:

Table 5.1: The area of each continuous region defined in each testcase.

Testcase\regions (unit sq.)	**t1**	**t2**	**t3**	**t4**	**t5**	**t6**	**t7**	**t8**
Collision-free	0.126	0.51	0.785	0.785	3.14	3.14	3.14	3.14
classify	0.66	2.638	11.775	27.475	75.36	703.36	958.5	1252.9
observe	77.715	75.8624	164.85	149.15	631.14	10597.5	14424.4	18840

1. There are 8 different testcases in varying sizes of continuous regions. Table 5.1 shows the area of the regions considered in each testcase. Observe that the areas tend to get larger as the testcase number increases.

2. Each testcase includes 9 discrete problem sets and 1 continuous problem set. Each discrete problem set is denoted as $pt(x)$, where x refers to the number of discrete waypoints defined per continuous region and it varies in between 1 to 40 (i.e. $x = \{1, 3, 5, 7, 10, 15, 20, 30, 40\}$). For instance, in the $pt(x = 3)$ problem set, the discrete

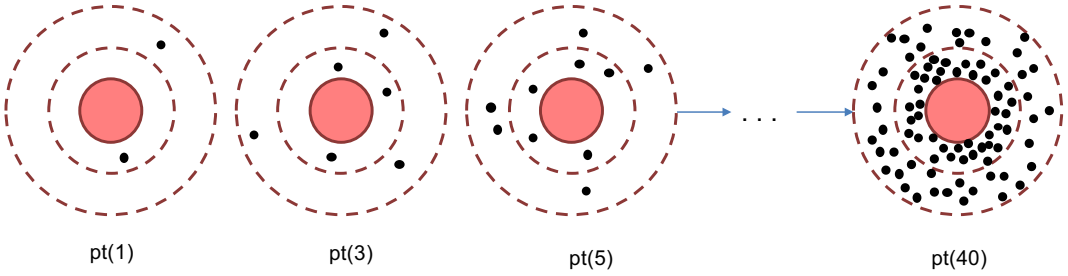


Figure 5.12: Illustration of defining varying number of discrete waypoints in regions.

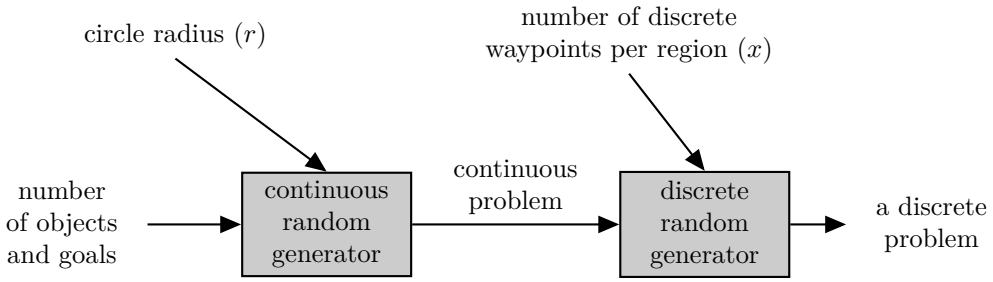


Figure 5.13: The process of continuous and discrete random problem generation.

random generator defines 3 discrete locations in each region. This is illustrated in Figure 5.12. The process of generating discrete and continuous problems is illustrated in Figure 5.13.

3. In each problem set, there are 20 problem instances in an increasing complexity order. The complexity increases in parallel to the number of classified object goals specified in each problem instance.

In order to compare the plan length and the plan quality, the duration of the move action is constrained by the Manhattan distance between locations for both discrete and continuous cases. In the case with continuous regions, the duration of the move action is constrained by linear functions, such as `(>= ?duration (+ (/ ?dy (maxV) ) (/ ?dx (maxV))))`, where `?dx`, `?dy` are control parameters representing the displacements in x- and y-directions. In the discrete case, it is precomputed by the problem generator (we explain the steps below) and it is constrained by `(= ?duration (distance ?w1 w2))` in the move action schema.

The problem instance generation for the continuous case works as follows. First, the continuous problem generator randomly specifies the coordinates of each object in configuration space. Second, the generator enumerates various numeric variables, such as `(observe_max_distance)` (see Appendix B.2.2), depending on the size of the region specified in the testcase. The numeric conditions encoded in the PDDL action schemas forms the feasible convex regions for each action.

In the discrete case, the discrete problem generator takes an already generated continuous problem instance and the discretisation granularity (i.e. x) as inputs. Then, it randomly generates x- and y- coordinates and their typed objects (x per region) that lies on each corresponding continuous region defined in the continuous problem. The generator computes the Manhattan distance between each coordinate pair and enumerates the `(distance`

`?w1 w2)` function for each location object combination. For instance, suppose that the coordinates of two discrete locations, `wp1` and `wp2`, are randomly defined as $(3, 3)$ and $(5, 5)$, respectively. The distance between these locations are computed and enumerated in the problem as `(= (distance wp1 wp2) 4)` (see Appendix B.1.2). Note that the coordinates are only considered for this computation and they are not encoded in PDDL.

5.4.2 Results

Having described the setup environment of the experiments, we present the results obtained in terms of solution quality, running time and the number of move actions found in the generated plan. We evaluate the planners on the observe-and-classify domain using a single core 3.4 GHz machine with 16 GB RAM (4GB allocated memory). Timeout was set to 30 minutes (1800 seconds).

5.4.2.1 Running Time

Figure 5.14 and Figure 5.15 illustrate the solution time comparison between planners. The data points positioned on the top edge of each graph (10000 seconds) represent that the planner was not able to find any solution to the problem instance within the allowed time. Note that the TFD and POPF planners use the same PDDL models and problem instances (as given in Appendix B.1). We can make the following comments on these results:

1. The running time of the discrete domain solutions (of POPF and TFD planners) tend to increase as the discretisation granularity increases. The planners are obliged to reason with an exponentially increasing number of combinatorials of typed objects when grounding the action schemas in parallel to slight increases in granularity. As a result, the search space exponentially grows due to increasing number of grounded actions. The effects of this concept was previously discussed in detail in Section 5.1.

2. Running domains with regions (marked as *popcorn*) take slightly longer time than the discrete ones. The rationale behind this result is that the POPCORN planner performs constraint space search (also known as C-state space search described in Section 3.5.5) and the forward state space search in parallel. The planner uses an external solver (in our case, it is LP solver) to reason with the accumulated constraints at every C-state. As discussed in Section 3.6, the overhead cost of using the LP is relatively higher than the consistency checking with the STN. Although the constraint space reasoning carries an overhead, the POPCORN planner catches up (and even performs faster) when the discretisation of waypoints gets refined (i.e. when granularity is greater than 15 waypoints), so the constraint space overhead is overtaken by the combinatorial blowup when many waypoints are modelled.

3. The solution time of the POPCORN planner is roughly in between the solution time of the POPF with the granularity of 10 and 15 waypoints; and the solution time of the TFD with the granularity of 1 and 3 waypoints per region. Then, we can say the TFD planner is recognisably slower than the POPF and POPCORN in most problem instances. This results from the fact that the TFD planner is based on multi-valued formalisation. It translates all propositional variables into multi-valued variable

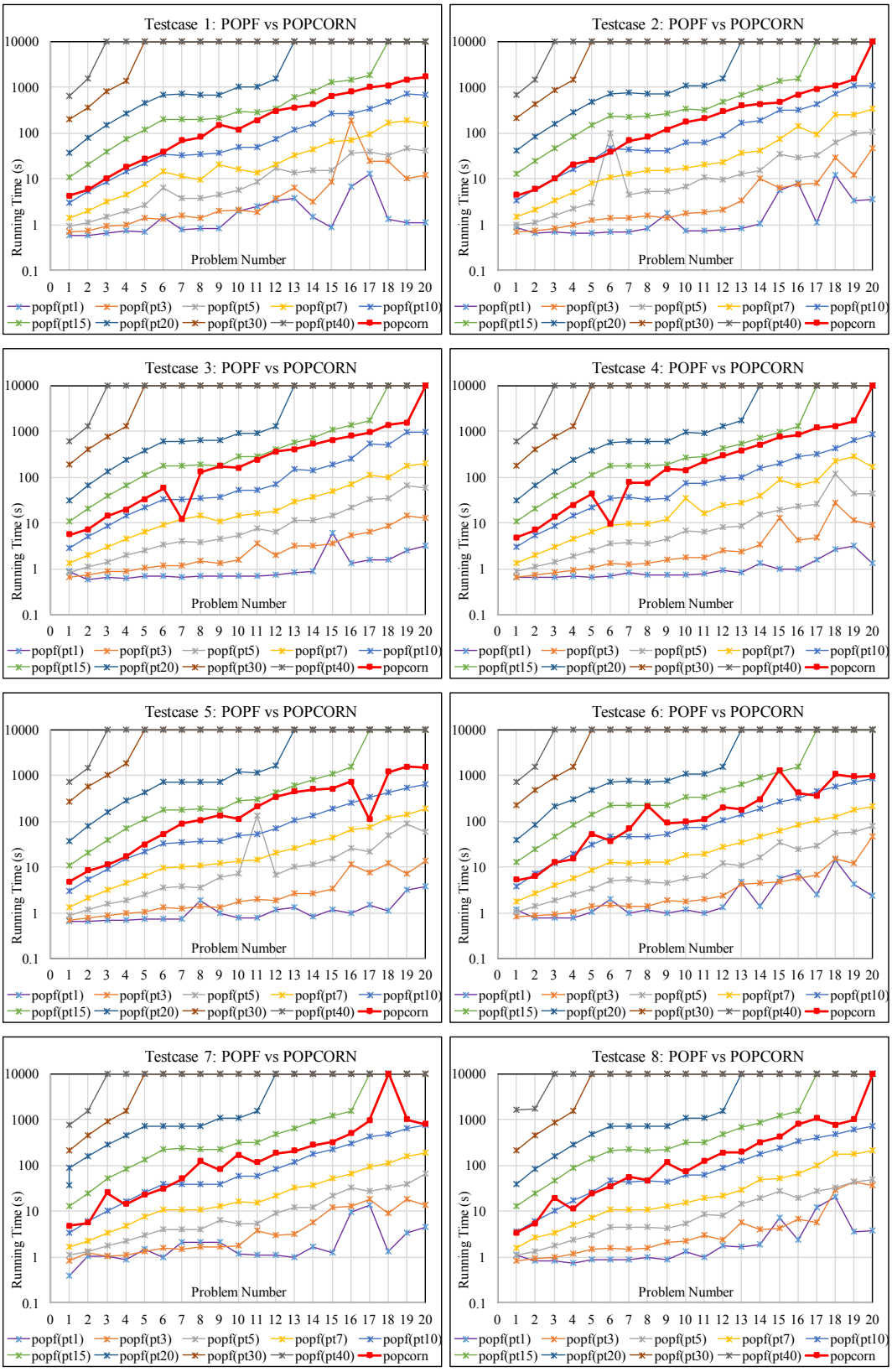


Figure 5.14: Running time comparison between POPF and POPCORN in 8 testcases.

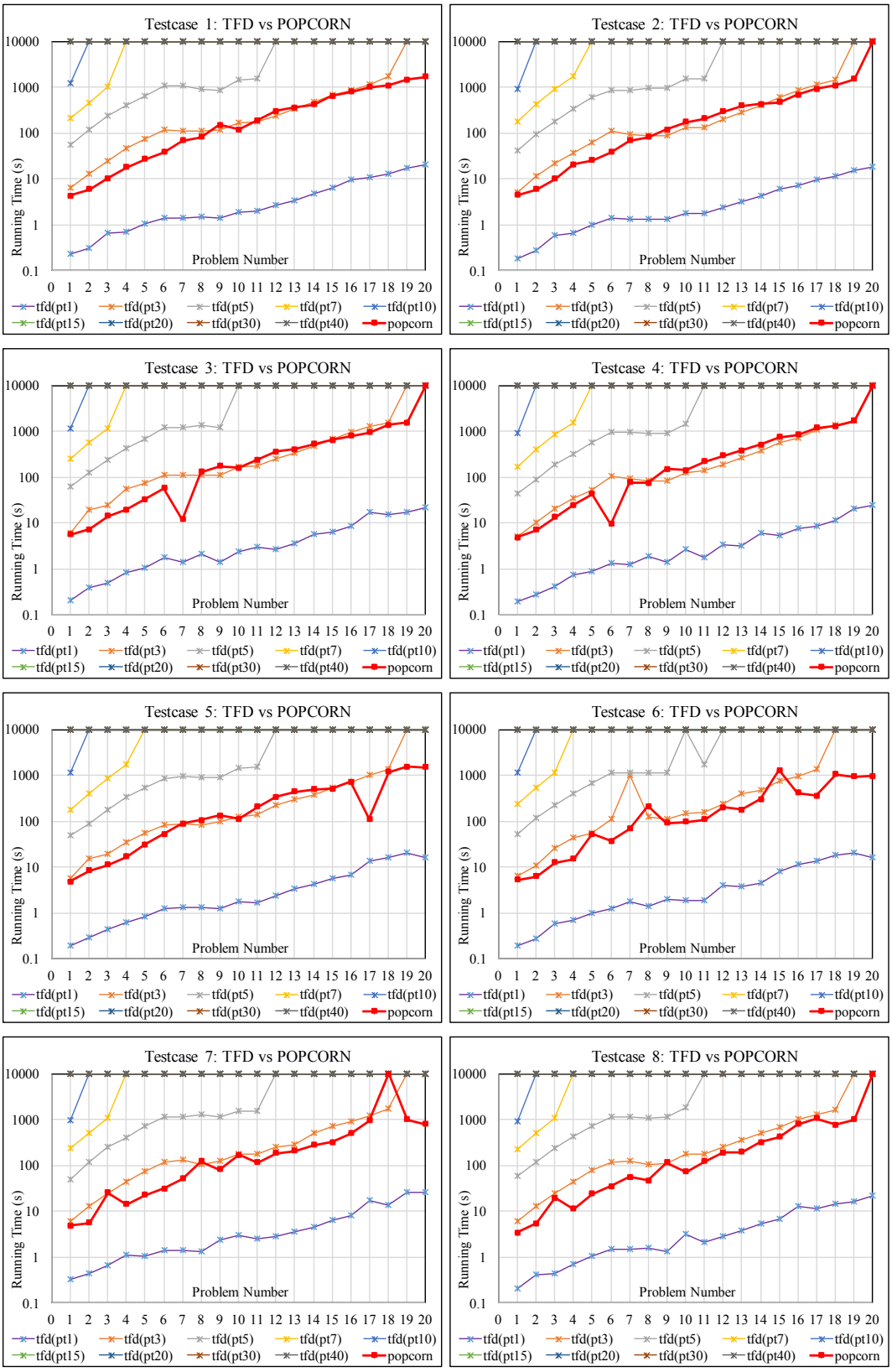


Figure 5.15: Running time comparison between TFD and POPCORN in 8 testcases.

encoding (i.e. *S*AS+) before it starts searching for a plan. This translation becomes highly cumbersome in problems requiring numerous propositional variables.

4. The effects of increasing the sizes of continuous regions (increasing the testcase number) on the running time was almost negligible in the discrete domain solutions (the results of POPF and TFD). However, we can observe a noticeable drop in solution time in continuous domain solutions from the testcase 5 and onward. We can comment on this as follows. As regions get larger, the overlap areas between the regions tend to increase. We discuss on this in depth later in this section.

5.4.2.2 Solution Quality

5.4.2.2.1 Makespan Figure 5.16 and Figure 5.17 illustrate the makespan comparison of the planners in all problem instances. The data points positioned on the top edge of each graph are the ones that the planners were not be able to find any solution to the problem instance within the time allowed. We can make the following comments on the results obtained:

1. The effects of increasing the sizes of continuous regions to the solution quality in discrete problem solutions is significant: the plan length tends to get larger as the regions become larger. This results from the fact that the random discretisation in larger regions increases the relative distance between discrete waypoints. Since the duration of the move action is constrained by the distance variable (as explained in Section 5.4.1), the durations of move actions tend to increase in discrete problems as the continuous regions become larger.

2. Although the variation in the solution qualities of the planners is significant, there is recognisable correlations between each other. For instance, the solution quality of POPCORN and TFD (the granularity of 1 and 3) in testcases 3 and 4 shows significant resemblance between each other. This result confirms that the Manhattan distance computations used in both discrete and continuous cases are indeed comparable attributes.

3. The POPCORN planner finds the shortest makespan solutions in most problem instances, because POPCORN optimises the durations of move actions that relies on control parameters. Even though POPF and TFD planners considers finding the shortest makespan, they are restricted to assign values to the duration parameters from the finitely enumerated distance computations provided in the problem instance. Thus, distance between discrete locations can be considered as an over-estimation to the shortest distances between regions.

4. Despite the fact that POPF can solve more problem instances than TFD, the solution quality difference between each other is considerable. It is worth mentioning that the scale in the makespan axes of the figures are different (2750 in POPF, 1250 in TFD). Thus, we can say that the TFD planner finds better solutions compared to the POPF planner.

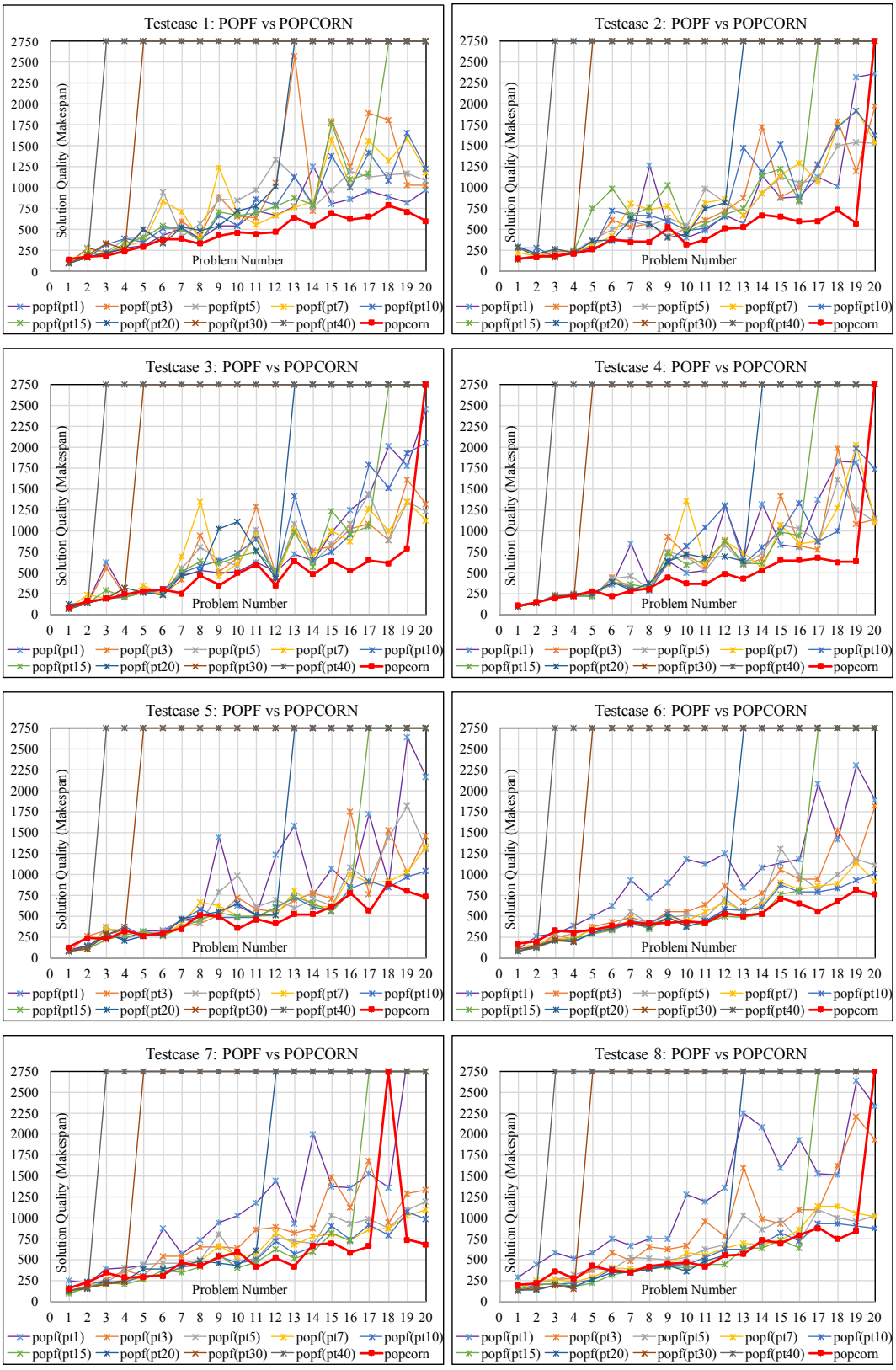


Figure 5.16: Makespan comparison between POPF and POPCORN in 8 testcases.

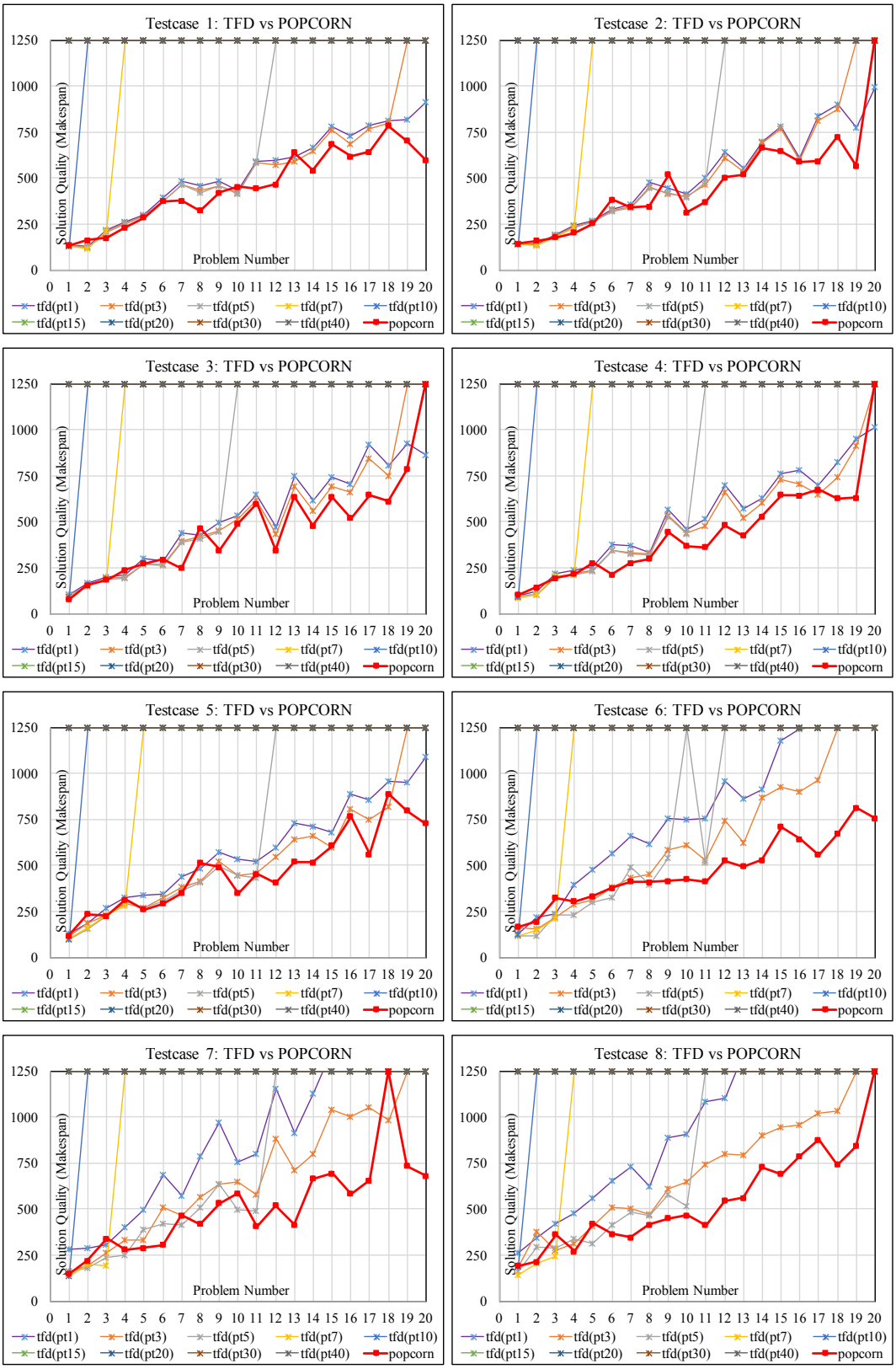


Figure 5.17: Makespan comparison between TFD and POPCORN in 8 testcases.

It is clear that constraint space reasoning produces shortest plans in almost all problem instances, giving incontrovertible evidence that control parameters make a real practical difference. Finding shortest plans is important in robot navigation as each move action can induce wear and tear on expensive robot components, then the use of control parameters could correspond to a significant financial saving over time.

5.4.2.2.2 Number of Move Actions From task planning perspective, the makespan is usually the main indicator of the solution quality. However, in robot navigation practice, it is desirable to obtain a direct trajectory than a trajectory with multiple layovers. We find the sum of move action occurrences in each plan generated in order to analyse the solution quality from action repetition perspective. Figure 5.18 and Figure 5.19 illustrates the number of move action per plan comparison between the planners. The data points lying on the top edge (where the number of move actions is equal to 80) of each figure indicates that the planner was not able to find a solution to this problem instance. We can make the following comments on this results:

1. We previously identified that the TFD planner produces plans with shorter makespan than the POPF planner does. However, on the contrary to this, one can easily notice that the plans generated by the TFD planner includes remarkably higher number of move actions than the ones by any other planner does. This behaviour is undesirable in our robot navigation scenario in the SQUIRREL project.

2. Adjusting the size of the continuous regions significantly affects the number of move actions in the plan. Observe that, in both figures, the number of move actions incrementally increases in plans generated by POPCORN in the first 5 testcases, but it sharply drops in testcases 6, 7 and 8. The rationale behind this is that once the regions become large enough, the regions start overlapping with each other. During navigation, the robot favours these overlapping regions to execute multiple activities from the same position. This result supports the idea of reducing excessive actions presented in Section 5.3 and illustrates the scenario in practice.

3. The plans generated by POPCORN (using continuous regions) consists of the least number of move actions compared to other planners. The reason behind this outcome is the use of different search techniques. In the discrete case, the planners have extremely high number of move action choices in the search space due to combinatorial groundings of this action schema. The following fact supports this statement: as the problem difficulty increases, the number of excessive move actions increases. In the continuous case, the number of move action choices is relatively low, where choosing a location is a matter of constraint satisfiability.

Table 5.2: The overall percentage of problems solved in each testcase.

Testcases:	**t1**	**t2**	**t3**	**t4**	**t5**	**t6**	**t7**	**t8**	**Avg.**
POPCORN	100%	95%	95%	95%	100%	100%	95%	95%	96.9%
POPF	75%	74.4%	75%	75%	74.4%	74.4%	74.4%	75%	74.7%
TFD	29%	30%	28.3%	30.0%	30%	28.3%	29.4%	28.9%	29.3%

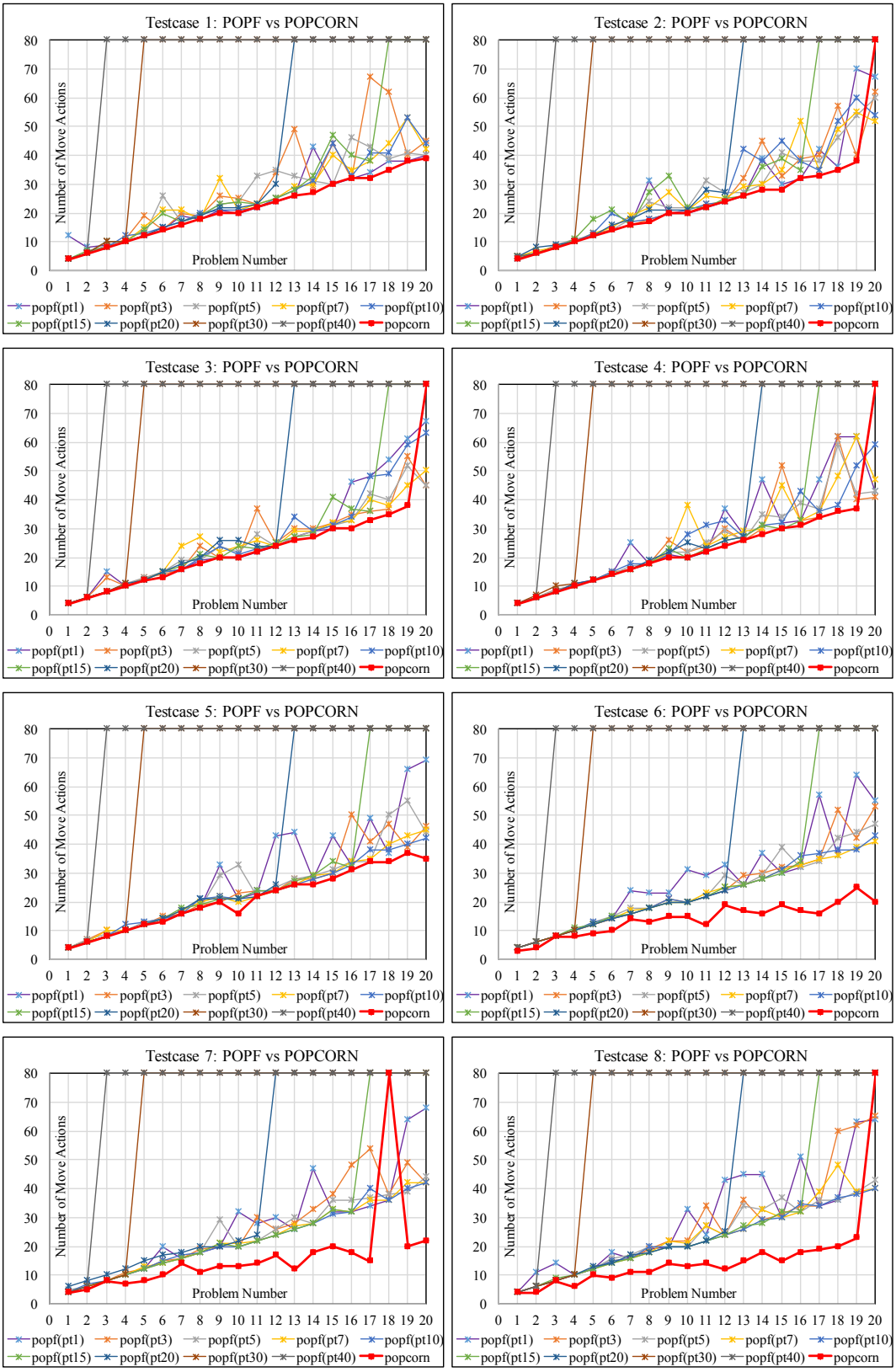


Figure 5.18: Number of move actions found in each plan (POPF and POPCORN).

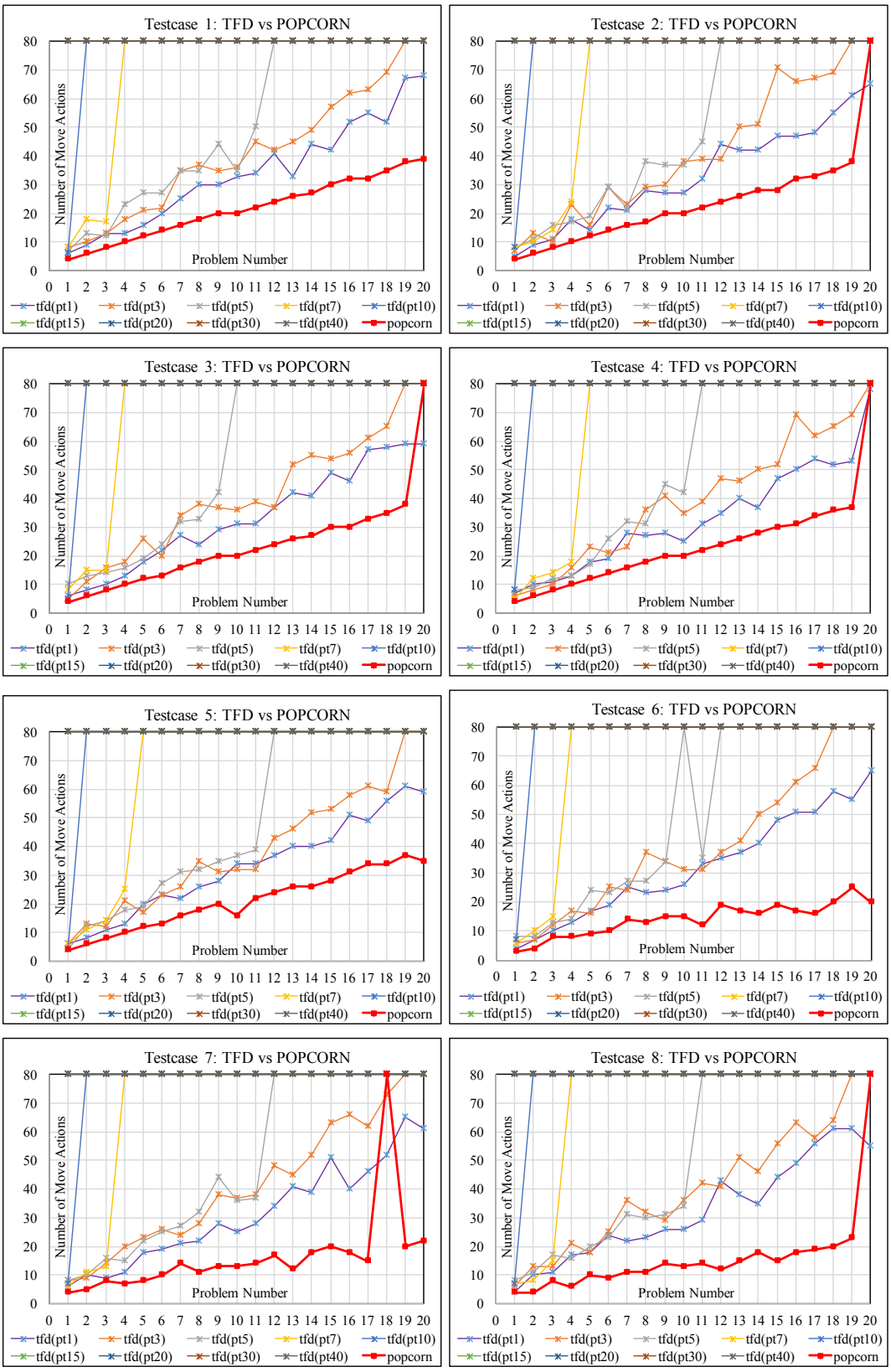


Figure 5.19: Number of move actions found in each plan (TFD and POPCORN).

5.4.2.3 Solvability

In this section, we talk about how reasoning with continuous convex regions affects the solvability compared to the discrete method. Table B.1.2 and Table 5.2 summarises the number of problem instances solved by each planner and their solvability in each testcase, respectively. Note that there are 20 problem instances in each problem set. Based on this table, we can remark the following statements:

1. In the discrete case, increasing the number of points defined in each region reduces the solvability of the planners. As discussed earlier, this is an expected outcome as action schema grounding is a combinatorial operation.

2. The results obtained in each testcase are somewhat identical. Then, we can say the effects of increasing the sizes of regions on the solvability is negligible.

3. The solvability of the TFD planner is extremely low compared to the other planners. As mentioned earlier, the TFD planner spends considerable time on translating the discrete variables into *SAS+* formula.

Table 5.3: The number of problem instances solved in each problem set (out of 20 instances).

	POPF									**POPCORN**
	pt1	**pt3**	**pt5**	**pt7**	**pt10**	**pt15**	**pt20**	**pt30**	**pt40**	**regions**
Testcase1	20	20	20	20	20	17	12	4	2	20
Testcase2	20	20	20	20	20	16	12	4	2	19
Testcase3	20	20	20	20	20	17	12	4	2	19
Testcase4	20	20	20	20	20	16	13	4	2	19
Testcase5	20	20	20	20	20	16	12	4	2	20
Testcase6	20	20	20	20	20	16	12	4	2	20
Testcase7	20	20	20	20	20	16	12	4	2	19
Testcase8	20	20	20	20	20	16	12	5	2	19
	TFD									
	pt1	**pt3**	**pt5**	**pt7**	**pt10**	**pt15**	**pt20**	**pt30**	**pt40**	
Testcase1	20	18	11	3	1	0	0	0	0	
Testcase2	20	18	11	4	1	0	0	0	0	
Testcase3	20	18	9	3	1	0	0	0	0	
Testcase4	20	19	10	4	1	0	0	0	0	
Testcase5	20	18	11	4	1	0	0	0	0	
Testcase6	20	17	10	3	1	0	0	0	0	
Testcase7	20	18	11	3	1	0	0	0	0	
Testcase8	20	18	10	3	1	0	0	0	0	

5.5 Summary

The action grounding operation can become a combinatorial problem in case the problem has considerable number of typed objects, and eventually the planning task can be easily impractical. In this chapter, we presented a scalable approach that radically changes the accustomed rules of the robot navigation in task planning. We propose representing the locations as continuous regions that avoids combinatorial problems, improves the solution quality and reflects the physical notions in a realistic way. Compared to the state-of-the-art

task and motion planning methods, our approach allows encoding geometric knowledge in expressive PDDL action level and it is domain-independent. We will consider integrating this approach with a motion planner in the future to analyse its behaviour in execution level.

Chapter 6

Conclusions

For more than a decade, the addition of time and numbers in automated planning has broadened its application area from solving only toy problems (e.g. Minesweeper, Solitaire) to real world problems; including production planning, space exploration missions and robotics. The control and optimisation of continuously executing dynamic processes is one of the main concerns of automated behaviour, however they are either ignored or considered as static by the off-the-shelf planning approaches. In this thesis, we have taken a further step towards solving realistic problems by introducing and investigating the notion of infinite domain action parameters (i.e. control parameters). We have introduced an extension to the standardised PDDL language to include these parameters, so that other researchers can benchmark their systems on this planning paradigm. Our language enables modelling different temporal-numeric planning problems with control parameters and it is not restricted to any specific application area (i.e. it enables modelling the problem in a domain-independent way). We have developed a temporal-numeric planner, POPCORN, that can reason with linear conditions and effects with multiple control parameters. POPCORN couples a forward state space heuristic search with a constraint space search to reason with both rich planning features (reasoning with time, numeric resources and causal links) and the dynamics of the environment (using control parameters). Although the existence of these parameters in planning as a search problem increases the branching factor, we have shown that (with a detailed set of experiments), coupling two search spaces (forwards search and constraint space search) is practical and efficient to find solutions for challenging planning problems.

POPF, the system that POPCORN was built on, has various planning capabilities, such as reasoning about the continuous numeric change, actions with flexible durations and required concurrency. Whilst POPCORN preserves the use of all of these features, it extends the system to reason about multiple numeric parameters that are kept flexible during planning (that are similar to the `?duration` parameter). We have shown that the use of `?duration` parameter is inadequate and unrealistic, as real-world applications can require *multiple* flexible parameters that need to be independent from the duration of the action. The cashpoint and the procurement domains are appropriate real-world examples, where the `?duration` parameter would not be a fitting surrogate for the control parameters.

We have identified that the continuous time relaxation (i.e. the infinity analysis) imposed

on a well-known temporal-numeric planning heuristic (the TRPG heuristic) undermines the basic informedness of planning problems with control parameters. We have proposed a novel relaxation technique, the refined infinity analysis, that combines the infinity analysis with an incremental graph expansion mechanism. We have implemented and tested this relaxation on the POPCORN planner (the resulting system is called POPCORN-R, which is POPCORN with Refined infinity analysis) and we have observed significant improvements on both the solvability and the solution quality of the planner on our benchmark domains, especially the ones with repetitive resource transfer via a set of producer-consumer actions. We have also shown that the heuristic scales and it is efficient in various planning problems with a large set of challenging instances.

We have exploited the proposed modelling technique in a robot navigation scenario that reinvents the representation of the waypoints on a 2-dimensional layout. It enables modelling waypoints as continuous regions rather than modelling them as discrete objects. This technique provides an abstract knowledge about the geometry of the environment to the task planner, so that it is more resilient to the unseen dynamics of the environment (the dynamics that are not included in the model, such as poor localisation due to sensor failure) than the models with discrete objects. We intend to support this statement by integrating our system in a planning and execution framework in the future (we provide the details in Section 6.2.3). We have observed recognisable improvements on the plan quality (i.e. the plans include less navigate actions) when the objects are relatively close to each other. We have also shown that it helps the planner to avoid combinatorial search problem, as it avoids grounding actions with location choices.

We summarise the contributions of the thesis by providing details of our methodology in the next section. We then describe the direction we intend to proceed in the future.

6.1 Summary of the Thesis

We have identified that the existing temporal-numeric planning approaches (their representations and modelling languages) pose a significant limitation on the dynamics of real world problems, in which their domains are restricted to finite values. We have proposed an extension to the most commonly used temporal-numeric planning language, so that we can encode multiple action specific flexible numeric variables. We have presented a planning approach (resulting in a system called POPCORN) handling problems modelled in this language and have shown that our approach can solve an interesting set of planning problems, where each carries different planning features (e.g. required concurrency, multiple or single control parameter per action) in a temporal setting. We compared POPCORN with our base system POPF, where it was offered a finite set of value options as a surrogate for control parameters. We observed that POPF was faster than POPCORN in most problem instances. The reason for this is that POPCORN uses a mathematical solver (i.e. LP solver) at every $C-$state to check the state consistency, whereas POPF solves these problems using only a STN. Although POPF was fast, it generated poor quality solutions (i.e. plan length, makespan) due to repetitions of the same actions. We have also benchmarked POPCORN with Scotty on domains in which the control parameters are modelled as rates. We observed that Scotty generates poor quality plans, because it makes early valuation choices (i.e.

finding fixed partial solutions during planning) on control parameters.

We have investigated extending the delete-relaxation based temporal-numeric heuristic of POPF, called the Temporal RPG, in two ways. The first one is a highly optimistic approach to estimating the bounds of control parameters and makes a minimal use of LP. It simply ignores the fact that the control parameter bounds can change during graph expansion. We observed that this extension is insufficient in numerically complex problems, in which numeric resources are repeatedly consumed and produced. We have identified that such problems were either unsolvable (due to dead-ends) or caused highly combinatorial search. A detailed observation has revealed that a relaxation method used in the heuristic, the infinity analysis, hinders realistic estimation of numeric bounds and require structural revision for tackling complex producer-consumer problems. We have proposed a new extension to the heuristic by modifying the infinity analysis, so that it can generate a hybrid (i.e. discrete and continuous) layered graph. We have also removed the use of the LP in this extension. We have implemented our approach in POPCORN, and called the new system POPCORN-R. We have benchmarked the systems POPCORN, POPCORN-R and cqScotty on a distinct set of domains and highly challenging numeric problems. When benchmarking POPCORN and POPCORN-R, we observed that our heuristic approach (i.e. POPCORN-R) has almost doubled the number of solved problems achieved by the planner, and significantly improved the solution time and quality. The comparison of our approach with cqScotty has revealed that the heuristic of cqScotty (that is a slightly modified version of the Temporal RPG) dramatically suffers from the numeric complexity of problems, and it could only solve half of them.

Lastly, we exploited the planning with control parameters approach in a task and motion planning scenario based on the SQUIRREL EU robotics project as a case study, in which the locations were modelled as continuous regions. The purpose of this study was to provide an estimated geometric information (of locations) to the task planner so that the solution quality could be improved and the potential combinatorial search problem can be avoided. In contrast to the existing task and motion planning integrations, our modelling technique can be handled in a domain-independent way. We have compared the continuous model with a simple discretised model using POPCORN, POPF and TFD planners on 8 different testcases (varying in size of regions). The experimental results have revealed that reasoning with continuous regions shows considerable improvement in the plan quality, especially when the regions become larger. The rationale behind this result is that the agent can execute multiple actions in overlapped continuous regions. On the other hand, this is not possible in models with discrete locations unless it is explicitly defined by the modeller. Although our modelling approach requires modelling and checking complex numeric constraints, it is still scalable as it reduces the search space size to a great extent.

Overall, the main contribution of the thesis is the introduction and management of integer and real-valued action parameters by giving choices off into mathematical programs (e.g. LP). Hence, by the separation of the combinatorial and numeric decision-making (while managing to maintain close integration between them), we achieve a sort of master-slave relationship between the two aspects by enforcing constraints on the combinatorial choices and exploiting search to reject these constraints when they cannot be satisfied. This yields a novel extension of the way that the duration parameter is already used in a few planners,

including COLIN and POPF.

6.2 Future Work

Our research summarised in the thesis has allowed us to discover a few possible directions for our prospective work. In this section, we briefly describe our next steps individually on the grounds of planning with control parameters.

6.2.1 Handling Quadratic Interactions in Domain Models

In this thesis, we restricted any possibility of non-linear interaction between variables and parameters as we intended to clarify the representation and the position of control parameters in automated planning. We believe that our research has come to a certain maturity to move forwards tackling problems with mathematical complexity, that is handling quadratic functions in pre- and post-conditions and effects of actions with existence of control parameter(s). We plan to investigate this work in two ways. First, we will examine numeric effects with control parameters in continuously evolving time (e.g. `(increase (x) (* ?v #t))`) in forwards heuristic search framework. The authors of Scotty and cqScotty have concentrated on this subject with two-staged planning process (i.e. finding a fixed plan followed by finding a flexible plan). In contrast to their work, we intend to keep the control parameters as duration-independent notions. This will allow us to solve far more realistic problems than their scope. For instance, filling a bathtub is a continuous process, in which the water flow rate is indeed independent from evolving time. We will make use of non-linear programming (NLP) solvers to handle the interaction.

Our second prospective work in this topic will be examining quadratic interactions between two control parameters, whose domains are independent from each other, in pre- and post-conditions and effects of actions. With the help of this feature, we could avoid the over-approximation technique of continuous regions used in Chapter 5, and even would have the possibility of defining different shaped regions (e.g. ellipses).

6.2.2 Extending the Heuristic to Handle Multiple Control Parameters in Resource Flow

We restricted the scope of the refined infinity analysis to handle numeric expressions with single numeric variable or a control parameter. In Remark 2, we have discussed the possibility of extending the capability of this approach to multi-variate numeric expressions by breaking them apart into individual flows. we will generalise the approach on the grounds of this idea. We expect that this work will require significant modifications to the $k-$limit and the numeric contribution computations. One possibility of achieving more robust estimates than our current algorithms is to create integer programming models (and to solve using MIP solver) to find the $k-$limit and the numeric contributions. Of course, this possibility will increase the overhead cost of the heuristic, but it will allow solving far more complex numeric problems than the ones we examined in this work. We will consider this prospective work for more generalised numeric planning problems (with or without control parameters).

6.2.3 Investigating the Execution Behaviour of Domains with Continuous Regions

We have presented a modelling approach in a robot navigation scenario, however we have not yet examined the behaviour in execution. We will integrate POPCORN with a the ROSPlan framework (Cashmore et al., 2015), which provides a generic way for planning and execution in a Robotic Operating System (ROS) (Quigley et al., 2009). ROSPlan can parse plans produced by POPF, and dispatches actions to be executed in ROS environment. We will first extend ROSPlan so that it can parse and dispatch plans with control parameters. We will modify the interface between the ROSPlan system and the generic motion planner of ROS, so that the motion planner can use the control parameter limits as estimated bounds.

Another challenge in this prospective work is to derive estimated continuous regions in domain models. In this thesis, the continuous regions were modelled manually and they were assumed to be collision-free regions. In reality, these regions should be constructed by the motion planner. We will work on automatically deriving (and updating) these regions based on feedback from the motion planner. Using learning approaches is a possibility to derive the most useful region estimates during execution.

6.3 List of Publications

1. Emre Savaş, Maria Fox, Derek Long, and Daniele Magazzeni. Planning Using Actions with Control Parameters. *In Proceedings of the Twenty-Second European Conference on Artificial Intelligence (ECAI 2016)*, 2016

2. Emre Ökkeş Savaş. Dissertation Abstract. *International Conference on Automated Planning and Scheduling Doctoral Consortium (ICAPS 2016)*, 2016

3. Emre Ökkeş Savaş, Maria Fox, Derek Long, and Daniele Magazzeni. Task Planning with Control Parameters. *In Proceedings of the 33rd Workshop of the UK Planning and Scheduling Special Interest Group (PlanSIG 2016)*, 2016

4. Michael Cashmore, Maria Fox, Derek Long, Daniele Magazzeni, Bram Ridder, Emre Savaş. ROSPlan: Planning in the Robot Operating System. *In Proceedings of the 6th Italian Workshop on Planning and Scheduling (IPS 2015)*, 2015

Appendices

Appendix A

The Relaxed Plan generated using Refined Infinity Analysis

Timestamps:	**Snap-actions:**	**# times applied** (i.e. *applyTimes*):
0.001:		
	(assemble_A), *start*	1
	(assemble_B), *start*	1
1.001:		
	(assemble_B), *end*	1
	(supply_raw_material E), *start*	∞
	(supply_raw_material F), *start*	∞
	(supply_raw_material G), *start*	∞
	(supply_raw_material H), *start*	∞
	(supply_raw_material I), *start*	∞
2.001:		
	(assemble_A), *end*	1
	(assemble_C), *start*	∞
	(assemble_D), *start*	∞
	(supply_raw_material E), *end*	∞
	(supply_raw_material F), *end*	∞
	(supply_raw_material G), *end*	∞
	(supply_raw_material H), *end*	∞
	(supply_raw_material I), *end*	∞
3.001:		
	(assemble_C), *end*	∞
	(assemble_D), *end*	∞
3.002:		
	(assemble_B), *start*	∞
4.002:		
	(assemble_B), *end*	∞
4.003:		
	(assemble_A), *start*	∞
6.003:		
	(assemble_A), *end*	∞

Appendix B

Observe-and-classify Domains

B.1 Locations as Discrete Notions Approach

B.1.1 The Domain Model

```
(define (domain squirrel-Discrete)
(:requirements :typing :durative-actions :fluents)
(:types location object)
(:predicates (empty ?loc - location) (ready)
    (observed ?obj - object) (classified ?obj - object)
    (robot_at ?from - location)
    (can_observe ?obj - object ?loc - location)
    (can_classify ?obj - object ?loc - location)
    (connected ?from ?to - location))
(:functions (distance ?from ?to - location) (maxV))

(:durative-action move_robot
:parameters(?from ?to - location)
:duration (and (= ?duration (/ (distance ?from ?to) (maxV))))
:condition (and (at start (connected ?from ?to))
    (at start (ready)) (at start (robot_at ?from)))
:effect (and  (at start (not (ready))) (at end (ready))
    (at start (not (robot_at ?from))) (at end (robot_at ?to))))

(:durative-action observe-object
:parameters(?loc - location ?obj - object)
:duration (and (= ?duration 5))
:condition (and (at start (ready)) (at start (empty ?loc))
    (at start (can_observe ?obj ?loc))
    (at start (robot_at ?loc)))
:effect (and (at start (not (ready)))
    (at end (ready)) (at end (observed ?obj))))

(:durative-action classify-object
:parameters(?loc - location ?obj - object)
```

```
:duration (and (= ?duration 5))
:condition (and (at start (ready))
    (at start (observed ?obj)) (at start (robot_at ?loc))
    (at start (can_classify ?obj ?loc)) (at start (empty ?loc)))
:effect (and (at start (not (ready))) (at end (ready))
    (at end (classified ?obj)))))
```

B.1.2 A problem Instance

```
(define (problem squirrel-problem0) (:domain squirrel-Discrete)
(:objects obj0 - object
o0observe0 o0observe1 o0observe2 - location
o0classify0 o0classify1 o0classify2 - location
initial_point - location)

(:init (ready) (robot_at initial_point) (= (maxV) 2)
(can_observe obj0 o0observe0) (empty initial_point) (empty o0classify2)
(can_observe obj0 o0observe1) (can_observe obj0 o0observe2)
(can_classify obj0 o0classify0) (can_classify obj0 o0classify1)
(can_classify obj0 o0classify2) (empty o0observe0) (empty o0observe1)
(empty o0observe2) (empty o0classify0) (empty o0classify1)
(connected initial_point o0observe0) (connected initial_point o0observe1)
(connected initial_point o0observe2) (connected initial_point o0classify0)
(connected initial_point o0classify1) (connected initial_point o0classify2)
(connected o0observe0 initial_point) (connected o0observe0 o0observe1)
(connected o0observe0 o0observe2) (connected o0observe0 o0classify0)
(connected o0observe0 o0classify1) (connected o0observe0 o0classify2)
(connected o0observe1 initial_point) (connected o0observe1 o0observe0)
(connected o0observe1 o0observe2) (connected o0observe1 o0classify0)
(connected o0observe1 o0classify1) (connected o0observe1 o0classify2)
(connected o0observe2 initial_point) (connected o0observe2 o0observe0)
(connected o0observe2 o0observe1) (connected o0observe2 o0classify0)
(connected o0observe2 o0classify1) (connected o0observe2 o0classify2)
(connected o0classify0 initial_point) (connected o0classify0 o0observe0)
(connected o0classify0 o0observe1) (connected o0classify0 o0observe2)
(connected o0classify0 o0classify1) (connected o0classify0 o0classify2)
(connected o0classify1 initial_point) (connected o0classify1 o0observe0)
(connected o0classify1 o0observe1) (connected o0classify1 o0observe2)
(connected o0classify1 o0classify0) (connected o0classify1 o0classify2)
(connected o0classify2 initial_point) (connected o0classify2 o0observe0)
(connected o0classify2 o0observe1) (connected o0classify2 o0observe2)
(connected o0classify2 o0classify0) (connected o0classify2 o0classify1)
(= (distance o0observe0 o0observe1) 2.12581)
(= (distance o0observe0 o0observe2) 3.8562)
(= (distance o0observe0 o0classify0) 1.83455)
(= (distance o0observe0 o0classify1) 1.58699)
(= (distance o0observe0 o0classify2) 1.55307)
(= (distance o0observe0 initial_point) 100.456)
```

```
(= (distance o0observe1 o0observe0) 2.12581)
(= (distance o0observe1 o0observe2) 2.51036)
(= (distance o0observe1 o0classify0) 0.29126)
(= (distance o0observe1 o0classify1) 0.538821)
(= (distance o0observe1 o0classify2) 0.683348)
(= (distance o0observe1 initial_point) 99.4494)
(= (distance o0observe2 o0observe0) 3.8562)
(= (distance o0observe2 o0observe1) 2.51036)
(= (distance o0observe2 o0classify0) 2.43234)
(= (distance o0observe2 o0classify1) 2.79487)
(= (distance o0observe2 o0classify2) 3.05718)
(= (distance o0observe2 initial_point) 101.96)
(= (distance o0classify0 o0observe0) 1.83455)
(= (distance o0classify0 o0observe1) 0.29126)
(= (distance o0classify0 o0observe2) 2.43234)
(= (distance o0classify0 o0classify1) 0.362535)
(= (distance o0classify0 o0classify2) 0.624845)
(= (distance o0classify0 initial_point) 99.5274)
(= (distance o0classify1 o0observe0) 1.58699)
(= (distance o0classify1 o0observe1) 0.538821)
(= (distance o0classify1 o0observe2) 2.79487)
(= (distance o0classify1 o0classify0) 0.362535)
(= (distance o0classify1 o0classify2) 0.26231)
(= (distance o0classify1 initial_point) 99.1649)
(= (distance o0classify2 o0observe0) 1.55307)
(= (distance o0classify2 o0observe1) 0.683348)
(= (distance o0classify2 o0observe2) 3.05718)
(= (distance o0classify2 o0classify0) 0.624845)
(= (distance o0classify2 o0classify1) 0.26231)
(= (distance o0classify2 initial_point) 98.9026)
(= (distance initial_point o0observe0) 100.456)
(= (distance initial_point o0observe1) 99.4494)
(= (distance initial_point o0observe2) 101.96)
(= (distance initial_point o0classify0) 99.5274)
(= (distance initial_point o0classify1) 99.1649)
(= (distance initial_point o0classify2) 98.9026))
(:goal (and (classified obj0)))
(:metric minimize (total-time)))
```

B.2 Locations as Continuous Regions Approach

B.2.1 The Domain Model

```
(define (domain squirrel)
(:requirements :typing :durative-actions :fluents :duration-inequalities)
(:types location object)
(:predicates (observed ?obj - object) (classified ?obj - object)
    (object_at ?obj - object ?loc - location) (ready)
```

```
    (empty_northwest ?loc - location) (empty_southwest ?loc - location)
    (empty_southeast ?loc - location) (empty_northeast ?loc - location))
(:functions (x_location ?loc - location) (y_location ?loc - location)
    (x_robot) (y_robot) (maxV) (classify_min_distance)
    (observe_min_distance) (classify_min_distance_times_sqrt2)
    (observe_min_distance_times_sqrt2) (observe_max_distance)
    (classify_max_distance_times_sqrt2) (classify_max_distance)
    (observe_max_distance_times_sqrt2))

(:durative-action move_robot
:parameters()
:control (?dx ?dy - number)
:duration (and (>= ?duration (/ ?dx (maxV)))
    (>= ?duration (/ (- ?dx) (maxV)))
    (>= ?duration (/ ?dy (maxV)))
    (>= ?duration (/ (- ?dy) (maxV)))
    (>= ?duration (+ (/ ?dy (maxV)) (/ ?dx (maxV))))
    (>= ?duration (+ (/ (- ?dy) (maxV)) (/ (- ?dx) (maxV))))
    (>= ?duration (+ (/ (- ?dy) (maxV)) (/ ?dx (maxV))))
    (>= ?duration (+ (/ ?dy (maxV)) (/ (- ?dx) (maxV))))
    (<= ?duration 200))
:condition (and (at start (ready))
    ;; the displacement of robot is inside [-100,100], [-100,100] square
    (at start (>= ?dx -100)) (at start (<= ?dx 100))
    (at start (>= ?dy -100)) (at start (<= ?dy 100))
    (at end (>= ?dx -100)) (at end (>= ?dy -100))
    (at end (<= ?dx 100)) (at end (<= ?dy 100))
    ; the robot must stay within [0, 100] [0, 100] square
    (over all (>= (x_robot) 0)) (over all (<= (x_robot) 100))
    (over all (>= (y_robot) 0)) (over all (<= (y_robot) 100)))
:effect (and (at start (not (ready))) (at end (ready))
    (at start (increase (x_robot) ?dx))
    (at start (increase (y_robot) ?dy))))

(:durative-action observe-object-Northeast
:parameters(?loc - location ?obj - object)
:duration (and (= ?duration 5))
:condition (and (at start (ready)) (at start (object_at ?obj ?loc))
    (at start (empty_northeast ?loc))
    (over all (>= (x_robot) 0)) (over all (<= (x_robot) 100))
    (over all (>= (y_robot) 0)) (over all (<= (y_robot) 100))
    (at start (> (x_robot) (x_location ?loc)))
    (at start (> (y_robot) (y_location ?loc)))
    (at start (<= (* 1 (- (x_robot) (x_location ?loc)))
        (observe_max_distance)))
    (at start (<= (* 1 (- (y_robot) (y_location ?loc)))
        (observe_max_distance)))
    (at start (<= (- (* 1 (- (x_robot) (x_location ?loc))) (*1
```

```
    (- (y_robot) (y_location ?loc)))) (observe_max_distance_times_sqrt2)))
    (at start (<= (+ (* 1 (- (x_robot) (x_location ?loc))) (*1
    (- (y_robot) (y_location ?loc)))) (observe_max_distance_times_sqrt2)))
    (at start (>= (* 1 (- (x_robot) (x_location ?loc)))
    (observe_min_distance)))
    (at start (>= (* 1 (- (y_robot) (y_location ?loc)))
    (observe_min_distance)))
    (at start (>= (- (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_robot) (y_location ?loc))))  (observe_min_distance_times_sqrt2)))
    (at start (>= (+ (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_robot) (y_location ?loc))))  (observe_min_distance_times_sqrt2)))
    (at end (<= (* 1 (- (x_robot) (x_location ?loc)))
        (observe_max_distance)))
    (at end (<= (* 1 (- (y_robot) (y_location ?loc)))
        (observe_max_distance)))
    (at end (<= (- (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_robot) (y_location ?loc))))  (observe_max_distance_times_sqrt2)))
    (at end (<= (+ (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_robot) (y_location ?loc))))  (observe_max_distance_times_sqrt2)))
    (at end (>= (* 1 (- (x_robot) (x_location ?loc)))
        (observe_min_distance)))
    (at end (>= (* 1 (- (y_robot) (y_location ?loc)))
        (observe_min_distance)))
    (at end (>= (- (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_robot) (y_location ?loc))))  (observe_min_distance_times_sqrt2)))
    (at end (>= (+ (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_robot) (y_location ?loc))))  (observe_min_distance_times_sqrt2)))
)
:effect (and (at start (not (ready))) (at end (ready))
    (at end (observed ?obj))))

(:durative-action observe-object-Northwest
:parameters(?loc - location ?obj - object)
:duration (and (= ?duration 5))
:condition (and (at start (ready)) (at start (object_at ?obj ?loc))
    (at start (empty_northwest ?loc))
    (over all (>= (x_robot) 0)) (over all (<= (x_robot) 100))
    (over all (>= (y_robot) 0)) (over all (<= (y_robot) 100))
    (at start (< (x_robot) (x_location ?loc)))
    (at start (> (y_robot) (y_location ?loc)))
    (at start (<= (* 1 (- (x_location ?loc) (x_robot)))
        (observe_max_distance)))
    (at start (<= (* 1 (- (y_robot) (y_location ?loc)))
        (observe_max_distance)))
    (at start (<= (- (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_robot) (y_location ?loc)))) (observe_max_distance_times_sqrt2)))
    (at start (<= (+ (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_robot) (y_location ?loc)))) (observe_max_distance_times_sqrt2)))
```

```
    (at start (>= (* 1 (- (x_location ?loc) (x_robot)))
        (observe_min_distance)))
    (at start (>= (* 1 (- (y_robot) (y_location ?loc)))
        (observe_min_distance)))
    (at start (>= (- (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_robot) (y_location ?loc)))) (observe_min_distance_times_sqrt2)))
    (at start (>= (+ (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_robot) (y_location ?loc)))) (observe_min_distance_times_sqrt2)))
    (at end (<= (* 1 (- (x_location ?loc) (x_robot)))
        (observe_max_distance)))
    (at end (<= (* 1 (- (y_robot) (y_location ?loc)))
        (observe_max_distance)))
    (at end (<= (- (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_robot) (y_location ?loc)))) (observe_max_distance_times_sqrt2)))
    (at end (<= (+ (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_robot) (y_location ?loc)))) (observe_max_distance_times_sqrt2)))
    (at end (>= (* 1 (- (x_location ?loc) (x_robot)))
        (observe_min_distance)))
    (at end (>= (* 1 (- (y_robot) (y_location ?loc)))
        (observe_min_distance)))
    (at end (>= (- (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_robot) (y_location ?loc)))) (observe_min_distance_times_sqrt2)))
    (at end (>= (+ (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_robot) (y_location ?loc)))) (observe_min_distance_times_sqrt2)))
)
:effect (and (at start (not (ready))) (at end (ready))
    (at end (observed ?obj))))

(:durative-action observe-object-southwest
:parameters(?loc - location ?obj - object)
:duration (and (= ?duration 5))
:condition (and (at start (ready)) (at start (object_at ?obj ?loc))
    (at start (empty_southwest ?loc))
    (over all (>= (x_robot) 0)) (over all (<= (x_robot) 100))
    (over all (>= (y_robot) 0)) (over all (<= (y_robot) 100))
    (at start (< (x_robot) (x_location ?loc)))
    (at start (< (y_robot) (y_location ?loc)))
    (at start (<= (* 1 (- (x_location ?loc) (x_robot)))
        (observe_max_distance)))
    (at start (<= (* 1 (- (y_location ?loc)(y_robot)))
        (observe_max_distance)))
    (at start (<= (- (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_location ?loc) (y_robot))))  (observe_max_distance_times_sqrt2)))
    (at start (<= (+ (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_location ?loc) (y_robot))))  (observe_max_distance_times_sqrt2)))
    (at start (>= (* 1 (- (x_location ?loc) (x_robot)))
        (observe_min_distance)))
    (at start (>= (* 1 (- (y_location ?loc) (y_robot)))
```

```
        (observe_min_distance)))
    (at start (>= (- (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_location ?loc) (y_robot))))  (observe_min_distance_times_sqrt2)))
    (at start (>= (+ (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_location ?loc) (y_robot))))  (observe_min_distance_times_sqrt2)))
    (at end (<= (* 1 (- (x_location ?loc) (x_robot)))
        (observe_max_distance)))
    (at end (<= (* 1 (- (y_location ?loc)(y_robot)))
        (observe_max_distance)))
    (at end (<= (- (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_location ?loc) (y_robot))))  (observe_max_distance_times_sqrt2)))
    (at end (<= (+ (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_location ?loc) (y_robot))))  (observe_max_distance_times_sqrt2)))
    (at end (>= (* 1 (- (x_location ?loc) (x_robot)))
        (observe_min_distance)))
    (at end (>= (* 1 (- (y_location ?loc) (y_robot)))
        (observe_min_distance)))
    (at end (>= (- (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_location ?loc) (y_robot)))) (observe_min_distance_times_sqrt2)))
    (at end (>= (+ (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_location ?loc) (y_robot)))) (observe_min_distance_times_sqrt2)))
)
:effect (and (at start (not (ready))) (at end (ready))
    (at end (observed ?obj))))

(:durative-action observe-object-southeast
:parameters(?loc - location ?obj - object)
:duration (and (= ?duration 5))
:condition (and (at start (ready)) (at start (object_at ?obj ?loc))
    (at start (empty_southeast ?loc))
    (over all (>= (x_robot) 0)) (over all (<= (x_robot) 100))
    (over all (>= (y_robot) 0)) (over all (<= (y_robot) 100))
    (at start (> (x_robot) (x_location ?loc)))
    (at start (< (y_robot) (y_location ?loc)))
    (at start (<= (* 1 (- (x_robot) (x_location ?loc)))
        (observe_max_distance)))
    (at start (<= (* 1 (- (y_location ?loc)(y_robot)))
        (observe_max_distance)))
    (at start (<= (- (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_location ?loc) (y_robot)))) (observe_max_distance_times_sqrt2)))
    (at start (<= (+ (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_location ?loc) (y_robot)))) (observe_max_distance_times_sqrt2)))
    (at start (>= (* 1 (- (x_robot) (x_location ?loc)))
        (observe_min_distance)))
    (at start (>= (* 1 (- (y_location ?loc) (y_robot)))
        (observe_min_distance)))
    (at start (>= (- (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_location ?loc) (y_robot)))) (observe_min_distance_times_sqrt2)))
```

```
    (at start (>= (+ (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_location ?loc) (y_robot)))) (observe_min_distance_times_sqrt2)))
    (at end (<= (* 1 (- (x_robot) (x_location ?loc)))
        (observe_max_distance)))
    (at end (<= (* 1 (- (y_location ?loc)(y_robot)))
        (observe_max_distance)))
    (at end (<= (- (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_location ?loc) (y_robot)))) (observe_max_distance_times_sqrt2)))
    (at end (<= (+ (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_location ?loc) (y_robot)))) (observe_max_distance_times_sqrt2)))
    (at end (>= (* 1 (- (x_robot) (x_location ?loc)))
        (observe_min_distance)))
    (at end (>= (* 1 (- (y_location ?loc) (y_robot)))
        (observe_min_distance)))
    (at end (>= (- (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_location ?loc) (y_robot)))) (observe_min_distance_times_sqrt2)))
    (at end (>= (+ (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_location ?loc) (y_robot)))) (observe_min_distance_times_sqrt2)))
)
:effect (and (at start (not (ready))) (at end (ready))
    (at end (observed ?obj))))

(:durative-action classify-object-Northeast
:parameters(?loc - location ?obj - object)
:duration (and (= ?duration 5))
:condition (and (at start (ready)) (at start (object_at ?obj ?loc))
    (at start (observed ?obj)) (at start (empty_northeast ?loc))
    (over all (>= (x_robot) 0)) (over all (<= (x_robot) 100))
    (over all (>= (y_robot) 0)) (over all (<= (y_robot) 100))
    (at start (> (x_robot) (x_location ?loc)))
    (at start (> (y_robot) (y_location ?loc)))
    (at start (<= (* 1 (- (x_robot) (x_location ?loc)))
        (classify_max_distance)))
    (at start (<= (* 1 (- (y_robot) (y_location ?loc)))
        (classify_max_distance)))
    (at start (<= (- (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_robot) (y_location ?loc)))) (classify_max_distance_times_sqrt2)))
    (at start (<= (+ (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_robot) (y_location ?loc)))) (classify_max_distance_times_sqrt2)))
    (at start (>= (* 1 (- (x_robot) (x_location ?loc)))
        (classify_min_distance)))
    (at start (>= (* 1 (- (y_robot) (y_location ?loc)))
        (classify_min_distance)))
    (at start (>= (- (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_robot) (y_location ?loc)))) (classify_min_distance_times_sqrt2)))
    (at start (>= (+ (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_robot) (y_location ?loc)))) (classify_min_distance_times_sqrt2)))
    (at end (<= (* 1 (- (x_robot) (x_location ?loc)))
```

```
        (classify_max_distance)))
    (at end (<= (* 1 (- (y_robot) (y_location ?loc)))
        (classify_max_distance)))
    (at end (<= (- (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_robot) (y_location ?loc)))) (classify_max_distance_times_sqrt2)))
    (at end (<= (+ (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_robot) (y_location ?loc)))) (classify_max_distance_times_sqrt2)))
    (at end (>= (* 1 (- (x_robot) (x_location ?loc)))
        (classify_min_distance)))
    (at end (>= (* 1 (- (y_robot) (y_location ?loc)))
        (classify_min_distance)))
    (at end (>= (- (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_robot) (y_location ?loc)))) (classify_min_distance_times_sqrt2)))
    (at end (>= (+ (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_robot) (y_location ?loc)))) (classify_min_distance_times_sqrt2)))
)
:effect (and (at start (not (ready))) (at end (ready))
    (at end (classified ?obj))))

(:durative-action classify-object-Northwest
:parameters(?loc - location ?obj - object)
:duration (and (= ?duration 5))
:condition (and (at start (ready)) (at start (object_at ?obj ?loc))
    (at start (observed ?obj)) (at start (empty_northwest ?loc))
    (over all (>= (x_robot) 0)) (over all (<= (x_robot) 100))
    (over all (>= (y_robot) 0)) (over all (<= (y_robot) 100))
    (at start (< (x_robot) (x_location ?loc)))
    (at start (> (y_robot) (y_location ?loc)))
    (at start (<= (* 1 (- (x_location ?loc) (x_robot)))
        (classify_max_distance)))
    (at start (<= (* 1 (- (y_robot) (y_location ?loc)))
        (classify_max_distance)))
    (at start (<= (- (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_robot) (y_location ?loc)))) (classify_max_distance_times_sqrt2)))
    (at start (<= (+ (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_robot) (y_location ?loc)))) (classify_max_distance_times_sqrt2)))
    (at start (>= (* 1 (- (x_location ?loc) (x_robot)))
        (classify_min_distance)))
    (at start (>= (* 1 (- (y_robot) (y_location ?loc)))
        (classify_min_distance)))
    (at start (>= (- (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_robot) (y_location ?loc)))) (classify_min_distance_times_sqrt2)))
    (at start (>= (+ (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_robot) (y_location ?loc)))) (classify_min_distance_times_sqrt2)))
    (at end (<= (* 1 (- (x_location ?loc) (x_robot)))
        (classify_max_distance)))
    (at end (<= (* 1 (- (y_robot) (y_location ?loc)))
        (classify_max_distance)))
```

```
    (at end (<= (- (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_robot) (y_location ?loc)))) (classify_max_distance_times_sqrt2)))
    (at end (<= (+ (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_robot) (y_location ?loc)))) (classify_max_distance_times_sqrt2)))
    (at end (>= (* 1 (- (x_location ?loc) (x_robot)))
        (classify_min_distance)))
    (at end (>= (* 1 (- (y_robot) (y_location ?loc)))
        (classify_min_distance)))
    (at end (>= (- (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_robot) (y_location ?loc)))) (classify_min_distance_times_sqrt2)))
    (at end (>= (+ (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_robot) (y_location ?loc)))) (classify_min_distance_times_sqrt2)))
)
:effect (and (at start (not (ready))) (at end (ready))
    (at end (classified ?obj))))

(:durative-action classify-object-southwest
:parameters(?loc - location ?obj - object)
:duration (and (= ?duration 5))
:condition (and (at start (ready)) (at start (object_at ?obj ?loc))
    (at start (observed ?obj)) (at start (empty_southwest ?loc))
    (over all (>= (x_robot) 0)) (over all (<= (x_robot) 100))
    (over all (>= (y_robot) 0)) (over all (<= (y_robot) 100))
    (at start (< (x_robot) (x_location ?loc)))
    (at start (< (y_robot) (y_location ?loc)))
    (at start (<= (* 1 (- (x_location ?loc) (x_robot)))
        (classify_max_distance)))
    (at start (<= (* 1 (- (y_location ?loc)(y_robot)))
        (classify_max_distance)))
    (at start (<= (- (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_location ?loc) (y_robot)))) (classify_max_distance_times_sqrt2)))
    (at start (<= (+ (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_location ?loc) (y_robot)))) (classify_max_distance_times_sqrt2)))
    (at start (>= (* 1 (- (x_location ?loc) (x_robot)))
        (classify_min_distance)))
    (at start (>= (* 1 (- (y_location ?loc) (y_robot)))
        (classify_min_distance)))
    (at start (>= (- (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_location ?loc) (y_robot)))) (classify_min_distance_times_sqrt2)))
    (at start (>= (+ (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_location ?loc) (y_robot)))) (classify_min_distance_times_sqrt2)))
    (at end (<= (* 1 (- (x_location ?loc) (x_robot)))
        (classify_max_distance)))
    (at end (<= (* 1 (- (y_location ?loc)(y_robot)))
        (classify_max_distance)))
    (at end (<= (- (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_location ?loc) (y_robot)))) (classify_max_distance_times_sqrt2)))
    (at end (<= (+ (* 1 (- (x_location ?loc) (x_robot))) (* 1
```

```
    (- (y_location ?loc) (y_robot)))) (classify_max_distance_times_sqrt2)))
    (at end (>= (* 1 (- (x_location ?loc) (x_robot)))
        (classify_min_distance)))
    (at end (>= (* 1 (- (y_location ?loc) (y_robot)))
        (classify_min_distance)))
    (at end (>= (- (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_location ?loc) (y_robot)))) (classify_min_distance_times_sqrt2)))
    (at end (>= (+ (* 1 (- (x_location ?loc) (x_robot))) (* 1
    (- (y_location ?loc) (y_robot)))) (classify_min_distance_times_sqrt2)))
)
:effect (and (at start (not (ready))) (at end (ready))
    (at end (classified ?obj))))

(:durative-action classify-object-southeast
:parameters(?loc - location ?obj - object)
:duration (and (= ?duration 5))
:condition (and (at start (ready)) (at start (object_at ?obj ?loc))
    (at start (observed ?obj)) (at start (empty_southeast ?loc))
    (over all (>= (x_robot) 0)) (over all (<= (x_robot) 100))
    (over all (>= (y_robot) 0)) (over all (<= (y_robot) 100))
    (at start (> (x_robot) (x_location ?loc)))
    (at start (< (y_robot) (y_location ?loc)))
    (at start (<= (* 1 (- (x_robot) (x_location ?loc)))
        (classify_max_distance)))
    (at start (<= (* 1 (- (y_location ?loc)(y_robot)))
        (classify_max_distance)))
    (at start (<= (- (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_location ?loc) (y_robot)))) (classify_max_distance_times_sqrt2)))
    (at start (<= (+ (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_location ?loc) (y_robot)))) (classify_max_distance_times_sqrt2)))
    (at start (>= (* 1 (- (x_robot) (x_location ?loc)))
        (classify_min_distance)))
    (at start (>= (* 1 (- (y_location ?loc) (y_robot)))
        (classify_min_distance)))
    (at start (>= (- (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_location ?loc) (y_robot)))) (classify_min_distance_times_sqrt2)))
    (at start (>= (+ (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_location ?loc) (y_robot)))) (classify_min_distance_times_sqrt2)))
    (at end (<= (* 1 (- (x_robot) (x_location ?loc)))
        (classify_max_distance)))
    (at end (<= (* 1 (- (y_location ?loc)(y_robot)))
        (classify_max_distance)))
    (at end (<= (- (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_location ?loc) (y_robot)))) (classify_max_distance_times_sqrt2)))
    (at end (<= (+ (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_location ?loc) (y_robot)))) (classify_max_distance_times_sqrt2)))
    (at end (>= (* 1 (- (x_robot) (x_location ?loc)))
        (classify_min_distance)))
```

```
    (at end (>= (* 1 (- (y_location ?loc) (y_robot)))
        (classify_min_distance)))
    (at end (>= (- (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_location ?loc) (y_robot)))) (classify_min_distance_times_sqrt2)))
    (at end (>= (+ (* 1 (- (x_robot) (x_location ?loc))) (* 1
    (- (y_location ?loc) (y_robot)))) (classify_min_distance_times_sqrt2)))
)
:effect (and (at start (not (ready))) (at end (ready))
    (at end (classified ?obj)))))
```

B.2.2 A problem Instance

```
(define (problem squirrel-problem0) (:domain squirrel)
(:objects obj0 - object  loc0 - location)

(:init (ready) (object_at obj0 loc0) (= (maxV) 2)
(= (classify_min_distance) 0.2) (= (classify_max_distance) 0.5)
(= (observe_min_distance) 0.5) (= (observe_max_distance) 5)
(= (classify_min_distance_times_sqrt2) 0.282843)
(= (classify_max_distance_times_sqrt2) 0.707107)
(= (observe_min_distance_times_sqrt2) 0.707107)
(= (observe_max_distance_times_sqrt2) 7.07107)
(= (x_robot) 0 ) (= (y_robot) 0 ) ;; initial_point
(= (x_location loc0) 70.736 ) (= (y_location loc0) 28.683))

(:goal (and (classified obj0)))
(:metric minimize (total-time)))
```

Bibliography

Ai-Chang, M., Bresina, J., Hsu, J., Jonsson, A., Kanefsky, B., McCurdy, M., Morris, P., Rajan, K., Vera, A., and Yglesias, J. Mapgen: Mixed-initiative activity planning for the mars exploration rover mission. In *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, 2004.

Aldinger, J., Mattmüller, R., and Göbelbecker, M. Complexity of interval relaxed numeric planning. In *Joint German/Austrian Conference on Artificial Intelligence (Künstliche Intelligenz)*, pages 19–31. Springer, 2015.

Bäckström, C. and Nebel, B. Complexity results for sas+ planning. *Computational Intelligence*, 11(4):625–655, 1995.

Bajada, J. *Temporal planning for rich numeric contexts.* PhD thesis, King's College London, 2016.

Bajada, J., Fox, M., and Long, D. Temporal planning with semantic attachment of non-linear monotonic continuous behaviours. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence (IJCAI 2015)*, pages 1523–1529, 2015.

Benton, J., Coles, A., and Coles, A. Temporal Planning with Preferences and Time-Dependent Continuous Costs. In *Proceedings of the Twenty-First International Conference on Automated Planning and Scheduling, ICAPS*, 2012.

Blum, A. L. and Furst, M. L. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1):281–300, 1997.

Bogomolov, S., Magazzeni, D., Podelski, A., and Wehrle, M. Planning as model checking in hybrid domains. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 2228–2234, 2014.

Bonet, B. and Geffner, H. Planning as heuristic search: New results. In *European Conference on Planning*, pages 360–372. Springer, 1999.

Bonet, B. and Geffner, H. Planning as heuristic search. *Artificial Intelligence*, 129(1-2): 5–33, 2001.

Bonet, B., Loerincs, G., and Geffner, H. A robust and fast action selection mechanism for planning. In *Proceedings of the fourteenth national conference on artificial intelligence and ninth conference on Innovative applications of artificial intelligence*

(AAAI'97/IAAI'97), pages 714–719, 1997. URL `https://www.aaai.org/Papers/Workshops/1997/WS-97-10/WS97-10-002.pdf`.

Botea, A., Enzenberger, M., Müller, M., and Schaeffer, J. Macro-FF: Improving AI Planning with Automatically Learned Macro-Operators. *Journal of Artificial Intelligence Research*, 24:581–621, 2005.

Bryce, D., Gao, S., Musliner, D. J., and Goldman, R. P. Smt-based nonlinear pddl+ planning. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, pages 3247–3253, 2015.

Bylander, T. The computational complexity of propositional strips planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

Cambon, S., Alami, R., and Gravot, F. A hybrid approach to intricate motion, manipulation and task planning. *The International Journal of Robotics Research*, 28(1):104–126, 2009.

Cashmore, M., Fox, M., Larkworthy, T., Long, D., and Magazzeni, D. Auv mission control via temporal planning. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*, pages 6535–6541. IEEE, 2014.

Cashmore, M., Fox, M., Long, D., Magazzeni, D., Ridder, B., Carrera, A., Palomeras, N., Hurtos, N., and Carreras, M. Rosplan: Planning in the robot operating system. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling*, pages 333–341, 2015.

Cashmore, M., Fox, M., Long, D., and Magazzeni, D. A compilation of the full pddl+ language into smt. In *Proceedings of the Twenty-Sixth International Conference on Automated Planning and Scheduling*, 2016.

Cimatti, A., Giunchiglia, E., Giunchiglia, F., and Traverso, P. Planning via model checking: A decision procedure for ar. *Recent Advances in AI planning*, pages 130–142, 1997.

Coles, A., Coles, A., Fox, M., and Long, D. "extending the use of inference in temporal planning as forwards search". In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, 2009a.

Coles, A., Coles, A., Fox, M., and Long, D. Temporal Planning in Domains with Linear Processes. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI 2009)*. AAAI Press, July 2009b.

Coles, A., Coles, A., Fox, M., and Long, D. Forward-Chaining Partial-Order Planning. In *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010)*, pages 42–49, 2010.

Coles, A., Coles, A., Fox, M., and Long, D. COLIN: Planning with Continuous Linear Numeric Change. *Journal of Artificial Intelligence Research (JAIR)*, pages 1–96, 2012.

Coles, A., Coles, A., Fox, M., and Long, D. A hybrid LP-RPG heuristic for modelling numeric resource flows in planning. *Journal of Artificial Intelligence Research*, 46:343–412, 2013.

Coles, A., Fox, M., and Smith, A. Online identification of useful macro-actions for planning. In *Proceedings of the International Conference on Automated Planning and Scheduling (ICAPS)*, pages 97–104, 2007.

Coles, A., Fox, M., Long, D., and Smith, A. Planning with Problems Requiring Temporal Coordination. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (AAAI 2008)*, July 2008a.

Coles, A., Fox, M., Long, D., and Smith, A. A Hybrid Relaxed Planning Graph-lp Heuristic for Numeric Planning Domains. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, pages 52–59, 2008b.

Coles, A., Fox, M., Halsey, K., Long, D., and Smith, A. Managing concurrency in temporal planning using planner-scheduler interaction. *Artificial Intelligence*, 173:1–44, 2009c.

Culberson, J. C. and Schaeffer, J. Pattern databases. *Computational Intelligence*, 14(3): 318–334, 1998.

Cushing, W., Kambhampati, S., Weld, D. S., et al. When is temporal planning really temporal? In *Proceedings of the 20th international joint conference on Artifical intelligence*, pages 1852–1859. Morgan Kaufmann Publishers Inc., 2007.

Dechter, R., Meiri, I., and Pearl, J. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, 1991.

Do, M. and Kambhampati, S. Sapa: A domain-independent heuristic metric temporal planner. In *Proceedings of the Sixth European Conference on Planning*, pages 82–91, 2014.

Do, M. B. and Kambhampati, S. Sapa: A multi-objective metric temporal planner. *Journal of Artificial Intelligence Research*, 20:155–194, 2003.

Dornhege, C., Eyerich, P., Keller, T., Trüg, S., Brenner, M., and Nebel, B. Semantic attachments for domain-independent planning systems. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, pages 114,121, 2009.

Edelkamp, S. Planning with pattern databases. In *Sixth European Conference on Planning*, pages 13–34, 2001.

Edelkamp, S. and Jabbar, S. Cost-optimal external planning. In *Proceedings of the National Conference on Artificial Intelligence*, volume 21, page 821. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

Edelkamp, S. and Schroedl, S. *Heuristic search - Theory and Applications*. Morgan Kaufmann, 2012.

Effinger, R. T., Williams, B. C., Kelly, G., and Sheehy, M. Dynamic Controllability of Temporally-flexible Reactive Programs. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, 2009.

Erdem, E., Haspalamutgil, K., Palaz, C., Patoglu, V., and Uras, T. Combining high-level causal reasoning with low-level geometric reasoning and motion planning for robotic manipulation. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*, pages 4575–4581. IEEE, 2011.

Erdogan, C. *Planning in constraint space for multi-body manipulation tasks*. PhD thesis, Georgia Institute of Technology, 2016.

Erdogan, C. and Stilman, M. Planning in constraint space: automated design of functional structures. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*, pages 1807–1812. IEEE, 2013.

Eyerich, P., Mattmüller, R., and Röger, G. Using the context-enhanced additive heuristic for temporal and numeric planning. *Towards Service Robots for Everyday Environments*, 76:49–64, 2012.

Fernández-González, E., Karpas, E., and Williams, B. C. Mixed Discrete-Continuous Heuristic Generative Planning Based on Flow Tubes. In *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence*, 2015a.

Fernández-González, E., Karpas, E., and Williams, B. C. Mixed Discrete-Continuous Heuristic Generative Planning Based on Flow Tubes. In *Proceedings of the 3rd Workshop on Planning and Robotics (PlanRob)*, pages 106–115, 2015b.

Fernández-González, E., Karpas, E., and Williams, B. C. Mixed discrete-continuous planning with convex optimization. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence (AAAI-17)*, pages 4574–4580, 2017.

Fikes, R. E. and Nilsson, N. J. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3-4):189–208, 1971.

Fox, M. and Long, D. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research (JAIR)*, 20:61–124, 2003.

Fox, M. and Long, D. Modelling mixed discrete-continuous domains for planning. *Journal of Artificial Intelligence Research*, 27:235–297, 2006.

Francès, G. and Geffner, H. Modeling and computation in planning: Better heuristics from more expressive languages. In *Proceedings of the Twenty-Fifth International Conference on Automated Planning and Scheduling*, pages 70–78, 2015.

Garrett, C. R., Lozano-Pérez, T., and Kaelbling, L. P. Ffrob: An efficient heuristic for task and motion planning. In *Algorithmic Foundations of Robotics XI*, pages 179–195. Springer, 2015.

Garrett, C. R., Lozano-Perez, T., and Kaelbling, L. P. Ffrob: Leveraging symbolic planning for efficient task and motion planning. *The International Journal of Robotics Research*, 37(1):104–136, 2018. URL `https://doi.org/10.1177/0278364917739114`.

Garrido, A., Onaindía, E., and Barber, F. A temporal planning system for time-optimal planning. In *Portuguese Conference on Artificial Intelligence*, pages 379–392. Springer, 2001.

Garrido, A., Fox, M., and Long, D. A temporal planning system for durative actions of pddl2.1. In *Proceedings of the 15th European Conference on Artificial Intelligence*, pages 586–590. IOS Press, 2002.

Geffner, H. and Bonet, B. A concise introduction to models and methods for automated planning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(1):1–141, 2013. URL `http://www.morganclaypool.com/doi/abs/10.2200/S00513ED1V01Y201306AIM022`.

Geffner, H. and Haslum, P. Admissible heuristics for optimal planning. In *Proceedings of the 5th Internat. Conf. of AI Planning Systems (AIPS 2000)*, pages 140–149, 2000.

Gerevini, A., Saetti, A., and Serina, I. An approach to temporal planning and scheduling in domains with predictable exogenous events. *Journal of Artificial Intelligence Research*, 25:187–231, 2006.

Gerevini, A. E., Saetti, A., and Serina, I. An approach to efficient planning with numerical fluents and multi-criteria plan quality. *Artificial Intelligence*, 172(8-9):899–944, 2008.

Gerevini, A. E., Haslum, P., Long, D., Saetti, A., and Dimopoulos, Y. Deterministic planning in the fifth international planning competition: Pddl3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5):619–668, 2009.

Ghallab, M. and Laruelle, H. Representation and control in ixtet, a temporal planner. In *Proceedings of the AIPS*, pages 61–67, 1994.

Ghallab, M., Nau, D., and Traverso, P. *Automated Planning Theory and Practice*. Morgan Kaufmann, 2004. ISBN 1-55860-856-7.

Gregory, P., Long, D., and Fox, M. Constraint Based Planning with Composable Substate Graphs. In *Proceedings of the Nineteenth European Conference on Artificial Intelligence (ECAI 2010)*, pages 453–458. IOS Press, 2010.

Hart, P. E., Nilsson, N. J., and Raphael, B. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2): 100–107, 1968.

Haslum, P. and Geffner, H. Heuristic planning with time and resources. In *Sixth European Conference on Planning*, 2001. URL `https://www.ida.liu.se/divisions/aiics/publications/ECP-2001-Heuristic-Planning-Time.pdf`.

Haslum, P., Botea, A., Helmert, M., Bonet, B., Koenig, S., et al. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, volume 7, pages 1007–1012, 2007.

Helmert, M. Decidability and undecidability results for planning with numerical state variables. In *Proceedings of the Sixth International Conference on Artificial Intelligence Planning and Scheduling (AIPS)*, 2002.

Helmert, M. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

Helmert, M., Haslum, P., Hoffmann, J., et al. Flexible abstraction heuristics for optimal sequential planning. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, pages 176–183, 2007.

Henzinger, T. A. The theory of hybrid automata. *Verification of Digital and Hybrid Systems*, pages 265–292, 2000.

Hoffmann, J. The Metric-FF Planning System: Translating 'Ignoring Delete Lists' to Numeric State Variables. *Journal of Artificial Intelligence Research*, pages 291–341, 2003.

Hoffmann, J. and Edelkamp, S. The deterministic part of ipc-4: An overview. *Journal of Artificial Intelligence Research*, 24:519–579, 2005.

Hoffmann, J. and Nebel, B. The ff planning system: Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.

Hoffmann, J., Porteous, J., and Sebastia, L. Ordered landmarks in planning. *Journal of Artificial Intelligence Research*, 22:215–278, 2004.

Hofmann, A. G. and Williams, B. C. Exploiting Spatial and Temporal Flexibility for Plan Execution for Hybrid, Under-actuated Robots. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, pages 948–955, 2006.

Howey, R. and Long, D. VAL's Progress : The Automatic Validation Tool for PDDL 2.1 used in The International Planning Competition. *Proceedings of the ICAPS Workshop on The Competition: Impact, Organization, Evaluation, Benchmarks*, 2003.

Huang, R., Chen, Y., and Zhang, W. An optimal temporally expressive planner: Initial results and application to p2p network optimization. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, 2009.

Ivankovic, F., Haslum, P., Thiébaux, S., Shivashankar, V., and Nau, D. S. Optimal planning with global numerical state constraints. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling*, 2014.

Karpas, E. and Domshlak, C. Cost-optimal planning with landmarks. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 1728–1733, 2009.

Keyder, E., Richter, S., and Helmert, M. Sound and complete landmarks for and/or graphs. In *Proceedings of the Nineteenth European Conference on Artificial Intelligence (ECAI 2010)*, volume 215, pages 335–340, 2010.

Kovacs, D. L. Bnf definition of pddl 3.1. *Unpublished manuscript from the IPC-2011 website*, 2011.

Kovacs, D. L. A multi-agent extension of pddl3.1. In *Proceedings of the Third Workshop on the International Planning Competition (IPC), ICAPS-2012, Atibaia, Brazil*, pages 19–27, 2012.

Laborie, P. Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. *Artificial Intelligence*, 143:151–188, 2003. URL `http://linkinghub.elsevier.com/retrieve/pii/S0004370202003624`.

Laborie, P. and Ghallab, M. Planning with sharable resource constraints. In *Proceedings of the Fourteenth international joint conference on Artificial intelligence*, volume 2, pages 1643–1649, 1995.

Léauté, T. and Williams, B. C. Coordinating agile systems through the model-based execution of temporal plans. In *Proceedings of the Twentieth National Conference on Artificial Intelligence*, pages 114–120, 2005.

Li, H. X. and Williams, B. C. Generative Planning for Hybrid Systems Based on Flow Tubes. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 2008)*, pages 206–213, 2008.

Lifschitz, V. On the semantics of strips. In *Reasoning about Actions and Plans: Proceedings of the 1986 Workshop*, pages 1–9, 1987.

Löhr, J., Eyerich, P., Winkler, S., and Nebel, B. Domain predictive control under uncertain numerical state information. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling*, 2013.

Long, D. and Fox, M. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research*, 20:1–59, 2003.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., and Wilkins, D. Pddl - the planning domain definition language. *Technical Report CVC TR98003/DCS TR1165. New Haven, CT*, 1998.

McDermott, D. M. The 1998 ai planning systems competition. *AI magazine*, 21(2):35, 2000.

Moore, R. E., Kearfott, R. B., and Cloud, M. J. *Introduction to interval analysis*. SIAM, 2009.

Muscettola, N., Nayak, P. P., Pell, B., and Williams, B. C. Remote agent: To boldly go where no ai system has gone before. *Artificial Intelligence*, 103(1-2):5–47, 1998.

Pantke, F., Edelkamp, S., and Herzog, O. Planning with Numeric Key Performance Indicators over Dynamic Organizations of Intelligent Agents. In *German Conference on Multi Agent System Technologies*, pages 138–155. Springer, 2014.

Pantke, F., Edelkamp, S., and Herzog, O. Symbolic Discrete-Time Planning with Continuous Numeric Action Parameters for Agent-controlled Processes. *Mechatronics*, 34:38–62, 2016.

Pearl, J. Heuristics. intelligent search strategies for computer problem solving. *The Addison-Wesley Series in Artificial Intelligence, Reading, Mass.: Addison-Wesley, 1985, Reprinted version*, 1985.

Pednault, E. P. Adl: Exploring the middle ground between strips and the situation calculus. *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, 89:324–332, 1989.

Penna, G. D., Magazzeni, D., Mercorio, F., and Intrigila, B. UPMurphi: A Tool for Universal Planning on PDDL+ Problems. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling*, 2009.

Piacentini, C., Alimisis, V., Fox, M., and Long, D. Combining a temporal planner with an external solver for the power balancing problem in an electricity network. In *Proceedings of the Twenty-Third International Conference on Automated Planning and Scheduling*, 2013.

Piotrowski, W., Fox, M., Long, D., Magazzeni, D., and Mercorio, F. Heuristic Planning for PDDL+ Domains. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI)*. AAAI Press, July 2016.

Plaku, E. and Le, D. Interactive search for action and motion planning with dynamics. *Journal of Experimental & Theoretical Artificial Intelligence*, 28(5):849–869, 2016.

Pohl, I. Heuristic search viewed as path finding in a graph. *Artificial intelligence*, 1(3-4): 193–204, 1970.

Porteous, J., Sebastia, L., and Hoffmann, J. On the extraction, ordering, and usage of landmarks in planning. In *6th European Conference on Planning*, 2001.

Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, 2009.

Reiter, R. On closed world data bases. *Logic and Data Bases*, 1977.

Richter, S. and Westphal, M. The lama planner: Guiding cost-based anytime planning with landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010.

Richter, S., Helmert, M., and Westphal, M. Landmarks revisited. In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, volume 8, pages 975–982, 2008.

Rintanen, J. et al. Complexity of concurrent temporal planning. In *Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling (ICAPS 2007)*, pages 280–287, 2007.

Russell, S. and Norvig, P. *Artificial intelligence a modern approach: international edition*. Pearson, 2010. ISBN 978-1292153964.

Sacerdoti, E. D. The nonlinear nature of plans. Technical report, Stanford Research Institute, 1975.

Sanner, S. Relational dynamic influence diagram language (rddl): Language description. *Unpublished ms. Australian National University*, page 32, 2010.

Savaş, E., Fox, M., Long, D., and Magazzeni, D. Planning Using Actions with Control Parameters. In *Proceedings of the Twenty-Second European Conference on Artificial Intelligence (ECAI 2016)*, pages 1185–1193. IOS Press, 2016.

Scala, E., Haslum, P., Thiebaux, S., and Ramirez, M. Interval-based relaxation for general numeric planning. In *Proceedings of the European Conference on Artificial Intelligence*, pages 655–663. IOS Press, 2016a.

Scala, E., Haslum, P., and Thiébaux, S. Heuristics for numeric planning via subgoaling. In *Proceedings of the International Joint Conferences on Artificial Intelligence*, pages 3228–3234, 2016b.

Scala, E., Haslum, P., Magazzeni, D., and Thiébaux, S. Landmarks for numeric planning problems. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, pages 4384–4390. AAAI Press, 2017.

Shin, J. A. and Davis, E. Processes and continuous change in a SAT-based planner. *Artificial Intelligence*, 166(1-2):194–253, 2005.

Smith, D. and Weld, D. Temporal planning with mutual exclusion reasoning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 326–333, 1999.

Srivastava, S., Fang, E., Riano, L., Chitnis, R., Russell, S., and Abbeel, P. Combined task and motion planning through an extensible planner-independent interface layer. In *Proceedings of the International Conference on Robotics and Automation (ICRA)*, pages 639–646. IEEE, 2014.

van den Briel, M., Benton, J., Kambhampati, S., and Vossen, T. An {LP}-Based Heuristic for Optimal Planning. In *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming*, pages 651–665, 2007.

van den Briel, M. H. and Kambhampati, S. Optiplan: Unifying ip-based and graph-based planning. *Journal of Artificial Intelligence Research*, 24:919–931, 2005.

Weld, D. S. An introduction to least commitment planning. *AI magazine*, 15(4):27–61, 1994.

Wolfman, S. A. and Weld, D. S. Combining linear programming and satisfiability solving for resource planning. *The Knowledge Engineering Review*, 16:85–99, 2001. ISSN 0269-8889.

Xie, F. *Exploration in Greedy Best-First Search for Satisficing Planning*. PhD thesis, University of Alberta, 2016.

Xie, F., Müller, M., Holte, R., and Imai, T. Type-based exploration with multiple search queues for satisficing planning. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 2395–2402, 2014.

Younes, H. L. and Littman, M. L. Ppddl1.0: An extension to pddl for expressing planning domains with probabilistic effects. *Technical Report CMU-CS-04-162*, 2004.

Younes, H. L. and Simmons, R. G. Vhpop: Versatile heuristic partial order planner. *Journal of Artificial Intelligence Research*, 20:405–430, 2003.

Young, R. C. The algebra of many-valued quantities. *Mathematische Annalen*, 104(1): 260–290, 1931.